

Beginner's Python Cheat Sheet

Variables and Strings

Variables are used to store values. A string is a series of characters, surrounded by single or double quotes.

Hello world

```
print("Hello world!")
```

Hello world with a variable

```
msg = "Hello world!"  
print(msg)
```

Concatenation (combining strings)

```
first_name = 'albert'  
last_name = 'einstein'  
full_name = first_name + ' ' + last_name  
print(full_name)
```

Lists

A list stores a series of items in a particular order. You access items using an index, or within a loop.

Make a list

```
bikes = ['trek', 'redline', 'giant']
```

Get the first item in a list

```
first_bike = bikes[0]
```

Get the last item in a list

```
last_bike = bikes[-1]
```

Looping through a list

```
for bike in bikes:  
    print(bike)
```

Adding items to a list

```
bikes = []  
bikes.append('trek')  
bikes.append('redline')  
bikes.append('giant')
```

Making numerical lists

```
squares = []  
for x in range(1, 11):  
    squares.append(x**2)
```

Lists (cont.)

List comprehensions

```
squares = [x**2 for x in range(1, 11)]
```

Slicing a list

```
finishers = ['sam', 'bob', 'ada', 'bea']  
first_two = finishers[:2]
```

Copying a list

```
copy_of_bikes = bikes[:]
```

Tuples

Tuples are similar to lists, but the items in a tuple can't be modified.

Making a tuple

```
dimensions = (1920, 1080)
```

If statements

If statements are used to test for particular conditions and respond appropriately.

Conditional tests

equals	x == 42
not equal	x != 42
greater than	x > 42
or equal to	x >= 42
less than	x < 42
or equal to	x <= 42

Conditional test with lists

```
'trek' in bikes  
'surly' not in bikes
```

Assigning boolean values

```
game_active = True  
can_edit = False
```

A simple if test

```
if age >= 18:  
    print("You can vote!")
```

If-elif-else statements

```
if age < 4:  
    ticket_price = 0  
elif age < 18:  
    ticket_price = 10  
else:  
    ticket_price = 15
```

Dictionaries

Dictionaries store connections between pieces of information. Each item in a dictionary is a key-value pair.

A simple dictionary

```
alien = {'color': 'green', 'points': 5}
```

Accessing a value

```
print("The alien's color is " + alien['color'])
```

Adding a new key-value pair

```
alien['x_position'] = 0
```

Looping through all key-value pairs

```
fav_numbers = {'eric': 17, 'ever': 4}  
for name, number in fav_numbers.items():  
    print(name + ' loves ' + str(number))
```

Looping through all keys

```
fav_numbers = {'eric': 17, 'ever': 4}  
for name in fav_numbers.keys():  
    print(name + ' loves a number')
```

Looping through all the values

```
fav_numbers = {'eric': 17, 'ever': 4}  
for number in fav_numbers.values():  
    print(str(number) + ' is a favorite')
```

User input

Your programs can prompt the user for input. All input is stored as a string.

Prompting for a value

```
name = input("What's your name? ")  
print("Hello, " + name + "!")
```

Prompting for numerical input

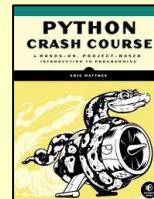
```
age = input("How old are you? ")  
age = int(age)
```

```
pi = input("What's the value of pi? ")  
pi = float(pi)
```

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



While loops

A *while loop* repeats a block of code as long as a certain condition is true.

A simple while loop

```
current_value = 1
while current_value <= 5:
    print(current_value)
    current_value += 1
```

Letting the user choose when to quit

```
msg = ''
while msg != 'quit':
    msg = input("What's your message? ")
    print(msg)
```

Functions

Functions are named blocks of code, designed to do one specific job. Information passed to a function is called an argument, and information received by a function is called a parameter.

A simple function

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

Passing an argument

```
def greet_user(username):
    """Display a personalized greeting."""
    print("Hello, " + username + "!")

greet_user('jesse')
```

Default values for parameters

```
def make_pizza(topping='bacon'):
    """Make a single-topping pizza."""
    print("Have a " + topping + " pizza!")
```

```
make_pizza()
make_pizza('pepperoni')
```

Returning a value

```
def add_numbers(x, y):
    """Add two numbers and return the sum."""
    return x + y

sum = add_numbers(3, 5)
print(sum)
```

Classes

A *class* defines the behavior of an object and the kind of information an object can store. The information in a class is stored in attributes, and functions that belong to a class are called methods. A child class inherits the attributes and methods from its parent class.

Creating a dog class

```
class Dog():
    """Represent a dog."""

    def __init__(self, name):
        """Initialize dog object."""
        self.name = name

    def sit(self):
        """Simulate sitting."""
        print(self.name + " is sitting.")

my_dog = Dog('Peso')

print(my_dog.name + " is a great dog!")
my_dog.sit()
```

Inheritance

```
class SARDog(Dog):
    """Represent a search dog."""

    def __init__(self, name):
        """Initialize the sardog."""
        super().__init__(name)

    def search(self):
        """Simulate searching."""
        print(self.name + " is searching.")

my_dog = SARDog('Willie')

print(my_dog.name + " is a search dog.")
my_dog.sit()
my_dog.search()
```

Infinite Skills

If you had infinite programming skills, what would you build?

As you're learning to program, it's helpful to think about the real-world projects you'd like to create. It's a good habit to keep an "ideas" notebook that you can refer to whenever you want to start a new project. If you haven't done so already, take a few minutes and describe three projects you'd like to create.

Working with files

Your programs can read from files and write to files. Files are opened in read mode ('r') by default, but can also be opened in write mode ('w') and append mode ('a').

Reading a file and storing its lines

```
filename = 'siddhartha.txt'
with open(filename) as file_object:
    lines = file_object.readlines()

for line in lines:
    print(line)
```

Writing to a file

```
filename = 'journal.txt'
with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
```

Appending to a file

```
filename = 'journal.txt'
with open(filename, 'a') as file_object:
    file_object.write("\nI love making games.")
```

Exceptions

Exceptions help you respond appropriately to errors that are likely to occur. You place code that might cause an error in the try block. Code that should run in response to an error goes in the except block. Code that should run only if the try block was successful goes in the else block.

Catching an exception

```
prompt = "How many tickets do you need? "
num_tickets = input(prompt)

try:
    num_tickets = int(num_tickets)
except ValueError:
    print("Please try again.")
else:
    print("Your tickets are printing.")
```

Zen of Python

Simple is better than complex

If you have a choice between a simple and a complex solution, and both work, use the simple solution. Your code will be easier to maintain, and it will be easier for you and others to build on that code later on.

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet - Lists

What are lists?

A list stores a series of items in a particular order. Lists allow you to store sets of information in one place, whether you have just a few items or millions of items. Lists are one of Python's most powerful features readily accessible to new programmers, and they tie together many important concepts in programming.

Defining a list

Use square brackets to define a list, and use commas to separate individual items in the list. Use plural names for lists, to make your code easier to read.

Making a list

```
users = ['val', 'bob', 'mia', 'ron', 'ned']
```

Accessing elements

Individual elements in a list are accessed according to their position, called the index. The index of the first element is 0, the index of the second element is 1, and so forth. Negative indices refer to items at the end of the list. To get a particular element, write the name of the list and then the index of the element in square brackets.

Getting the first element

```
first_user = users[0]
```

Getting the second element

```
second_user = users[1]
```

Getting the last element

```
newest_user = users[-1]
```

Modifying individual items

Once you've defined a list, you can change individual elements in the list. You do this by referring to the index of the item you want to modify.

Changing an element

```
users[0] = 'valerie'  
users[-2] = 'ronald'
```

Adding elements

You can add elements to the end of a list, or you can insert them wherever you like in a list.

Adding an element to the end of the list

```
users.append('amy')
```

Starting with an empty list

```
users = []  
users.append('val')  
users.append('bob')  
users.append('mia')
```

Inserting elements at a particular position

```
users.insert(0, 'joe')  
users.insert(3, 'bea')
```

Removing elements

You can remove elements by their position in a list, or by the value of the item. If you remove an item by its value, Python removes only the first item that has that value.

Deleting an element by its position

```
del users[-1]
```

Removing an item by its value

```
users.remove('mia')
```

Popping elements

If you want to work with an element that you're removing from the list, you can "pop" the element. If you think of the list as a stack of items, pop() takes an item off the top of the stack. By default pop() returns the last element in the list, but you can also pop elements from any position in the list.

Pop the last item from a list

```
most_recent_user = users.pop()  
print(most_recent_user)
```

Pop the first item in a list

```
first_user = users.pop(0)  
print(first_user)
```

List length

The len() function returns the number of items in a list.

Find the length of a list

```
num_users = len(users)  
print("We have " + str(num_users) + " users.")
```

Sorting a list

The sort() method changes the order of a list permanently. The sorted() function returns a copy of the list, leaving the original list unchanged. You can sort the items in a list in alphabetical order, or reverse alphabetical order. You can also reverse the original order of the list. Keep in mind that lowercase and uppercase letters may affect the sort order.

Sorting a list permanently

```
users.sort()
```

Sorting a list permanently in reverse alphabetical order

```
users.sort(reverse=True)
```

Sorting a list temporarily

```
print(sorted(users))  
print(sorted(users, reverse=True))
```

Reversing the order of a list

```
users.reverse()
```

Looping through a list

Lists can contain millions of items, so Python provides an efficient way to loop through all the items in a list. When you set up a loop, Python pulls each item from the list one at a time and stores it in a temporary variable, which you provide a name for. This name should be the singular version of the list name.

The indented block of code makes up the body of the loop, where you can work with each individual item. Any lines that are not indented run after the loop is completed.

Printing all items in a list

```
for user in users:  
    print(user)
```

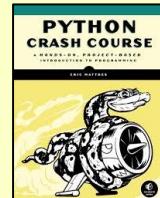
Printing a message for each item, and a separate message afterwards

```
for user in users:  
    print("Welcome, " + user + "!")  
  
print("Welcome, we're glad to see you all!")
```

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



The range() function

You can use the `range()` function to work with a set of numbers efficiently. The `range()` function starts at 0 by default, and stops one number below the number passed to it. You can use the `list()` function to efficiently generate a large list of numbers.

Printing the numbers 0 to 1000

```
for number in range(1001):
    print(number)
```

Printing the numbers 1 to 1000

```
for number in range(1, 1001):
    print(number)
```

Making a list of numbers from 1 to a million

```
numbers = list(range(1, 1000001))
```

Simple statistics

There are a number of simple statistics you can run on a list containing numerical data.

Finding the minimum value in a list

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
youngest = min(ages)
```

Finding the maximum value

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
oldest = max(ages)
```

Finding the sum of all values

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
total_years = sum(ages)
```

Slicing a list

You can work with any set of elements from a list. A portion of a list is called a slice. To slice a list start with the index of the first item you want, then add a colon and the index after the last item you want. Leave off the first index to start at the beginning of the list, and leave off the last index to slice through the end of the list.

Getting the first three items

```
finishers = ['kai', 'abe', 'ada', 'gus', 'zoe']
first_three = finishers[:3]
```

Getting the middle three items

```
middle_three = finishers[1:4]
```

Getting the last three items

```
last_three = finishers[-3:]
```

Copying a list

To copy a list make a slice that starts at the first item and ends at the last item. If you try to copy a list without using this approach, whatever you do to the copied list will affect the original list as well.

Making a copy of a list

```
finishers = ['kai', 'abe', 'ada', 'gus', 'zoe']
copy_of_finishers = finishers[:]
```

List comprehensions

You can use a loop to generate a list based on a range of numbers or on another list. This is a common operation, so Python offers a more efficient way to do it. List comprehensions may look complicated at first; if so, use the for loop approach until you're ready to start using comprehensions.

To write a comprehension, define an expression for the values you want to store in the list. Then write a for loop to generate input values needed to make the list.

Using a loop to generate a list of square numbers

```
squares = []
for x in range(1, 11):
    square = x**2
    squares.append(square)
```

Using a comprehension to generate a list of square numbers

```
squares = [x**2 for x in range(1, 11)]
```

Using a loop to convert a list of names to upper case

```
names = ['kai', 'abe', 'ada', 'gus', 'zoe']

upper_names = []
for name in names:
    upper_names.append(name.upper())
```

Using a comprehension to convert a list of names to upper case

```
names = ['kai', 'abe', 'ada', 'gus', 'zoe']

upper_names = [name.upper() for name in names]
```

Styling your code

Readability counts

- Use four spaces per indentation level.
- Keep your lines to 79 characters or fewer.
- Use single blank lines to group parts of your program visually.

Tuples

A tuple is like a list, except you can't change the values in a tuple once it's defined. Tuples are good for storing information that shouldn't be changed throughout the life of a program. Tuples are designated by parentheses instead of square brackets. (You can overwrite an entire tuple, but you can't change the individual elements in a tuple.)

Defining a tuple

```
dimensions = (800, 600)
```

Looping through a tuple

```
for dimension in dimensions:
    print(dimension)
```

Overwriting a tuple

```
dimensions = (800, 600)
print(dimensions)
```

```
dimensions = (1200, 900)
```

Visualizing your code

When you're first learning about data structures such as lists, it helps to visualize how Python is working with the information in your program. pythontutor.com is a great tool for seeing how Python keeps track of the information in a list. Try running the following code on pythontutor.com, and then run your own code.

Build a list and print the items in the list

```
dogs = []
dogs.append('willie')
dogs.append('hootz')
dogs.append('peso')
dogs.append('goblin')

for dog in dogs:
    print("Hello " + dog + "!")
print("I love these dogs!")
```

```
print("\nThese were my first two dogs:")
old_dogs = dogs[:2]
for old_dog in old_dogs:
    print(old_dog)
```

```
del dogs[0]
dogs.remove('peso')
print(dogs)
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Dictionaries

What are dictionaries?

Python's dictionaries allow you to connect pieces of related information. Each piece of information in a dictionary is stored as a key-value pair. When you provide a key, Python returns the value associated with that key. You can loop through all the key-value pairs, all the keys, or all the values.

Defining a dictionary

Use curly braces to define a dictionary. Use colons to connect keys and values, and use commas to separate individual key-value pairs.

Making a dictionary

```
alien_0 = {'color': 'green', 'points': 5}
```

Accessing values

To access the value associated with an individual key give the name of the dictionary and then place the key in a set of square brackets. If the key you're asking for is not in the dictionary, an error will occur.

You can also use the `get()` method, which returns `None` instead of an error if the key doesn't exist. You can also specify a default value to use if the key is not in the dictionary.

Getting the value associated with a key

```
alien_0 = {'color': 'green', 'points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

Getting the value with `get()`

```
alien_0 = {'color': 'green'}

alien_color = alien_0.get('color')
alien_points = alien_0.get('points', 0)

print(alien_color)
print(alien_points)
```

Adding new key-value pairs

You can store as many key-value pairs as you want in a dictionary, until your computer runs out of memory. To add a new key-value pair to an existing dictionary give the name of the dictionary and the new key in square brackets, and set it equal to the new value.

This also allows you to start with an empty dictionary and add key-value pairs as they become relevant.

Adding a key-value pair

```
alien_0 = {'color': 'green', 'points': 5}

alien_0['x'] = 0
alien_0['y'] = 25
alien_0['speed'] = 1.5
```

Adding to an empty dictionary

```
alien_0 = {}
alien_0['color'] = 'green'
alien_0['points'] = 5
```

Modifying values

You can modify the value associated with any key in a dictionary. To do so give the name of the dictionary and enclose the key in square brackets, then provide the new value for that key.

Modifying values in a dictionary

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)
```

```
# Change the alien's color and point value.
alien_0['color'] = 'yellow'
alien_0['points'] = 10
print(alien_0)
```

Removing key-value pairs

You can remove any key-value pair you want from a dictionary. To do so use the `del` keyword and the dictionary name, followed by the key in square brackets. This will delete the key and its associated value.

Deleting a key-value pair

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

del alien_0['points']
print(alien_0)
```

Visualizing dictionaries

Try running some of these examples on pythontutor.com.

Looping through a dictionary

You can loop through a dictionary in three ways: you can loop through all the key-value pairs, all the keys, or all the values.

A dictionary only tracks the connections between keys and values; it doesn't track the order of items in the dictionary. If you want to process the information in order, you can sort the keys in your loop.

Looping through all key-value pairs

```
# Store people's favorite languages.
fav_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
}
```

```
# Show each person's favorite language.
for name, language in fav_languages.items():
    print(name + ": " + language)
```

Looping through all the keys

```
# Show everyone who's taken the survey.
for name in fav_languages.keys():
    print(name)
```

Looping through all the values

```
# Show all the languages that have been chosen.
for language in fav_languages.values():
    print(language)
```

Looping through all the keys in order

```
# Show each person's favorite language,
# in order by the person's name.
for name in sorted(fav_languages.keys()):
    print(name + ": " + language)
```

Dictionary length

You can find the number of key-value pairs in a dictionary.

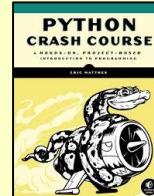
Finding a dictionary's length

```
num_responses = len(fav_languages)
```

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



Nesting — A list of dictionaries

It's sometimes useful to store a set of dictionaries in a list; this is called nesting.

Storing dictionaries in a list

```
# Start with an empty list.  
users = []  
  
# Make a new user, and add them to the list.  
new_user = {  
    'last': 'fermi',  
    'first': 'enrico',  
    'username': 'efermi',  
}  
users.append(new_user)  
  
# Make another new user, and add them as well.  
new_user = {  
    'last': 'curie',  
    'first': 'marie',  
    'username': 'mcurie',  
}  
users.append(new_user)  
  
# Show all information about each user.  
for user_dict in users:  
    for k, v in user_dict.items():  
        print(k + ":" + v)  
    print("\n")
```

You can also define a list of dictionaries directly, without using `append()`:

```
# Define a list of users, where each user  
#   is represented by a dictionary.  
users = [  
    {  
        'last': 'fermi',  
        'first': 'enrico',  
        'username': 'efermi',  
    },  
    {  
        'last': 'curie',  
        'first': 'marie',  
        'username': 'mcurie',  
    },  
]  
  
# Show all information about each user.  
for user_dict in users:  
    for k, v in user_dict.items():  
        print(k + ":" + v)  
    print("\n")
```

Nesting — Lists in a dictionary

Storing a list inside a dictionary allows you to associate more than one value with each key.

Storing lists in a dictionary

```
# Store multiple languages for each person.  
fav_languages = {  
    'jen': ['python', 'ruby'],  
    'sarah': ['c'],  
    'edward': ['ruby', 'go'],  
    'phil': ['python', 'haskell'],  
}  
  
# Show all responses for each person.  
for name, langs in fav_languages.items():  
    print(name + ":")  
    for lang in langs:  
        print("- " + lang)
```

Nesting — A dictionary of dictionaries

You can store a dictionary inside another dictionary. In this case each value associated with a key is itself a dictionary.

Storing dictionaries in a dictionary

```
users = {  
    'aeinstein': {  
        'first': 'albert',  
        'last': 'einstein',  
        'location': 'princeton',  
    },  
    'mcurie': {  
        'first': 'marie',  
        'last': 'curie',  
        'location': 'paris',  
    },  
}  
  
for username, user_dict in users.items():  
    print("\nUsername: " + username)  
    full_name = user_dict['first'] + " "  
    full_name += user_dict['last']  
    location = user_dict['location']  
  
    print("\tFull name: " + full_name.title())  
    print("\tLocation: " + location.title())
```

Levels of nesting

Nesting is extremely useful in certain situations. However, be aware of making your code overly complex. If you're nesting items much deeper than what you see here there are probably simpler ways of managing your data, such as using classes.

Using an OrderedDict

Standard Python dictionaries don't keep track of the order in which keys and values are added; they only preserve the association between each key and its value. If you want to preserve the order in which keys and values are added, use an `OrderedDict`.

Preserving the order of keys and values

```
from collections import OrderedDict  
  
# Store each person's languages, keeping  
#   track of who responded first.  
fav_languages = OrderedDict()  
  
fav_languages['jen'] = ['python', 'ruby']  
fav_languages['sarah'] = ['c']  
fav_languages['edward'] = ['ruby', 'go']  
fav_languages['phil'] = ['python', 'haskell']  
  
# Display the results, in the same order they  
#   were entered.  
for name, langs in fav_languages.items():  
    print(name + ":")  
    for lang in langs:  
        print("- " + lang)
```

Generating a million dictionaries

You can use a loop to generate a large number of dictionaries efficiently, if all the dictionaries start out with similar data.

A million aliens

```
aliens = []  
  
# Make a million green aliens, worth 5 points  
#   each. Have them all start in one row.  
for alien_num in range(1000000):  
    new_alien = {}  
    new_alien['color'] = 'green'  
    new_alien['points'] = 5  
    new_alien['x'] = 20 * alien_num  
    new_alien['y'] = 0  
    aliens.append(new_alien)  
  
# Prove the list contains a million aliens.  
num.aliens = len(aliens)
```

```
print("Number of aliens created:")  
print(num.aliens)
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — If Statements and While Loops

What are if statements? What are while loops?

If statements allow you to examine the current state of a program and respond appropriately to that state. You can write a simple if statement that checks one condition, or you can create a complex series of if statements that identify the exact conditions you're looking for.

While loops run as long as certain conditions remain true. You can use while loops to let your programs run as long as your users want them to.

Conditional Tests

A conditional test is an expression that can be evaluated as True or False. Python uses the values True and False to decide whether the code in an if statement should be executed.

Checking for equality

A single equal sign assigns a value to a variable. A double equal sign (==) checks whether two values are equal.

```
>>> car = 'bmw'  
>>> car == 'bmw'  
True  
>>> car = 'audi'  
>>> car == 'bmw'  
False
```

Ignoring case when making a comparison

```
>>> car = 'Audi'  
>>> car.lower() == 'audi'  
True
```

Checking for inequality

```
>>> topping = 'mushrooms'  
>>> topping != 'anchovies'  
True
```

Numerical comparisons

Testing numerical values is similar to testing string values.

Testing equality and inequality

```
>>> age = 18  
>>> age == 18  
True  
>>> age != 18  
False
```

Comparison operators

```
>>> age = 19  
>>> age < 21  
True  
>>> age <= 21  
True  
>>> age > 21  
False  
>>> age >= 21  
False
```

Checking multiple conditions

You can check multiple conditions at the same time. The and operator returns True if all the conditions listed are True. The or operator returns True if any condition is True.

Using and to check multiple conditions

```
>>> age_0 = 22  
>>> age_1 = 18  
>>> age_0 >= 21 and age_1 >= 21  
False  
>>> age_1 = 23  
>>> age_0 >= 21 and age_1 >= 21  
True
```

Using or to check multiple conditions

```
>>> age_0 = 22  
>>> age_1 = 18  
>>> age_0 >= 21 or age_1 >= 21  
True  
>>> age_0 = 18  
>>> age_0 >= 21 or age_1 >= 21  
False
```

Boolean values

A boolean value is either True or False. Variables with boolean values are often used to keep track of certain conditions within a program.

Simple boolean values

```
game_active = True  
can_edit = False
```

If statements

Several kinds of if statements exist. Your choice of which to use depends on the number of conditions you need to test. You can have as many elif blocks as you need, and the else block is always optional.

Simple if statement

```
age = 19
```

```
if age >= 18:  
    print("You're old enough to vote!")
```

If-else statements

```
age = 17
```

```
if age >= 18:  
    print("You're old enough to vote!")  
else:  
    print("You can't vote yet.")
```

The if-elif-else chain

```
age = 12
```

```
if age < 4:  
    price = 0  
elif age < 18:  
    price = 5  
else:  
    price = 10  
  
print("Your cost is $" + str(price) + ".")
```

Conditional tests with lists

You can easily test whether a certain value is in a list. You can also test whether a list is empty before trying to loop through the list.

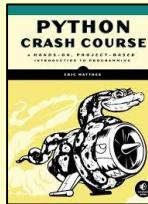
Testing if a value is in a list

```
>>> players = ['al', 'bea', 'cyn', 'dale']  
>>> 'al' in players  
True  
>>> 'eric' in players  
False
```

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



Conditional tests with lists (cont.)

Testing if a value is not in a list

```
banned_users = ['ann', 'chad', 'dee']
user = 'erin'

if user not in banned_users:
    print("You can play!")
```

Checking if a list is empty

```
players = []

if players:
    for player in players:
        print("Player: " + player.title())
else:
    print("We have no players yet!")
```

Accepting input

You can allow your users to enter input using the `input()` statement. In Python 3, all input is stored as a string.

Simple input

```
name = input("What's your name? ")
print("Hello, " + name + ".")
```

Accepting numerical input

```
age = input("How old are you? ")
age = int(age)

if age >= 18:
    print("\nYou can vote!")
else:
    print("\nYou can't vote yet.")
```

Accepting input in Python 2.7

Use `raw_input()` in Python 2.7. This function interprets all input as a string, just as `input()` does in Python 3.

```
name = raw_input("What's your name? ")
print("Hello, " + name + ".")
```

While loops

A while loop repeats a block of code as long as a condition is True.

Counting to 5

```
current_number = 1

while current_number <= 5:
    print(current_number)
    current_number += 1
```

While loops (cont.)

Letting the user choose when to quit

```
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program.

message = ""
while message != 'quit':
    message = input(prompt)

    if message != 'quit':
        print(message)
```

Using a flag

```
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program.

active = True
while active:
    message = input(prompt)
```

```
    if message == 'quit':
        active = False
    else:
        print(message)
```

Using break to exit a loop

```
prompt = "\nWhat cities have you visited?"
prompt += "\nEnter 'quit' when you're done.

while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        print("I've been to " + city + "!")
```

Accepting input with Sublime Text

Sublime Text doesn't run programs that prompt the user for input. You can use Sublime Text to write programs that prompt for input, but you'll need to run these programs from a terminal.

Breaking out of loops

You can use the `break` statement and the `continue` statement with any of Python's loops. For example you can use `break` to quit a for loop that's working through a list or a dictionary. You can use `continue` to skip over certain items when looping through a list or dictionary as well.

While loops (cont.)

Using continue in a loop

```
banned_users = ['eve', 'fred', 'gary', 'helen']

prompt = "\nAdd a player to your team."
prompt += "\nEnter 'quit' when you're done.

players = []
while True:
    player = input(prompt)
    if player == 'quit':
        break
    elif player in banned_users:
        print(player + " is banned!")
        continue
    else:
        players.append(player)

print("\nYour team:")
for player in players:
    print(player)
```

Avoiding infinite loops

Every while loop needs a way to stop running so it won't continue to run forever. If there's no way for the condition to become False, the loop will never stop running.

An infinite loop

```
while True:
    name = input("\nWho are you? ")
    print("Nice to meet you, " + name + "!")
```

Removing all instances of a value from a list

The `remove()` method removes a specific value from a list, but it only removes the first instance of the value you provide. You can use a while loop to remove all instances of a particular value.

Removing all cats from a list of pets

```
pets = ['dog', 'cat', 'dog', 'fish', 'cat',
        'rabbit', 'cat']

print(pets)

while 'cat' in pets:
    pets.remove('cat')

print(pets)
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Functions

What are functions?

Functions are named blocks of code designed to do one specific job. Functions allow you to write code once that can then be run whenever you need to accomplish the same task. Functions can take in the information they need, and return the information they generate. Using functions effectively makes your programs easier to write, read, test, and fix.

Defining a function

The first line of a function is its definition, marked by the keyword `def`. The name of the function is followed by a set of parentheses and a colon. A docstring, in triple quotes, describes what the function does. The body of a function is indented one level.

To call a function, give the name of the function followed by a set of parentheses.

Making a function

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

Passing information to a function

Information that's passed to a function is called an argument; information that's received by a function is called a parameter. Arguments are included in parentheses after the function's name, and parameters are listed in parentheses in the function's definition.

Passing a single argument

```
def greet_user(username):
    """Display a simple greeting."""
    print("Hello, " + username + "!")

greet_user('jesse')
greet_user('diana')
greet_user('brandon')
```

Positional and keyword arguments

The two main kinds of arguments are positional and keyword arguments. When you use positional arguments Python matches the first argument in the function call with the first parameter in the function definition, and so forth.

With keyword arguments, you specify which parameter each argument should be assigned to in the function call. When you use keyword arguments, the order of the arguments doesn't matter.

Using positional arguments

```
def describe_pet(animal, name):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")
```

```
describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')
```

Using keyword arguments

```
def describe_pet(animal, name):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")
```

```
describe_pet(animal='hamster', name='harry')
describe_pet(name='willie', animal='dog')
```

Default values

You can provide a default value for a parameter. When function calls omit this argument the default value will be used. Parameters with default values must be listed after parameters without default values in the function's definition so positional arguments can still work correctly.

Using a default value

```
def describe_pet(name, animal='dog'):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")
```

```
describe_pet('harry', 'hamster')
describe_pet('willie')
```

Using None to make an argument optional

```
def describe_pet(animal, name=None):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    if name:
        print("Its name is " + name + ".")
```

```
describe_pet('hamster', 'harry')
describe_pet('snake')
```

Return values

A function can return a value or a set of values. When a function returns a value, the calling line must provide a variable in which to store the return value. A function stops running when it reaches a `return` statement.

Returning a single value

```
def get_full_name(first, last):
    """Return a neatly formatted full name."""
    full_name = first + ' ' + last
    return full_name.title()
```

```
musician = get_full_name('jimi', 'hendrix')
print(musician)
```

Returning a dictionary

```
def build_person(first, last):
    """Return a dictionary of information about a person."""
    person = {'first': first, 'last': last}
    return person
```

```
musician = build_person('jimi', 'hendrix')
print(musician)
```

Returning a dictionary with optional values

```
def build_person(first, last, age=None):
    """Return a dictionary of information about a person."""
    person = {'first': first, 'last': last}
    if age:
        person['age'] = age
    return person
```

```
musician = build_person('jimi', 'hendrix', 27)
print(musician)

musician = build_person('janis', 'joplin')
print(musician)
```

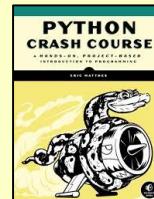
Visualizing functions

Try running some of these examples on pythontutor.com.

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



Passing a list to a function

You can pass a list as an argument to a function, and the function can work with the values in the list. Any changes the function makes to the list will affect the original list. You can prevent a function from modifying a list by passing a copy of the list as an argument.

Passing a list as an argument

```
def greet_users(names):
    """Print a simple greeting to everyone."""
    for name in names:
        msg = "Hello, " + name + "!"
        print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

Allowing a function to modify a list

The following example sends a list of models to a function for printing. The original list is emptied, and the second list is filled.

```
def print_models(unprinted, printed):
    """3d print a set of models."""
    while unprinted:
        current_model = unprinted.pop()
        print("Printing " + current_model)
        printed.append(current_model)

# Store some unprinted designs,
# and print each of them.
unprinted = ['phone case', 'pendant', 'ring']
printed = []
print_models(unprinted, printed)

print("\nUnprinted:", unprinted)
print("Printed:", printed)
```

Preventing a function from modifying a list

The following example is the same as the previous one, except the original list is unchanged after calling `print_models()`.

```
def print_models(unprinted, printed):
    """3d print a set of models."""
    while unprinted:
        current_model = unprinted.pop()
        print("Printing " + current_model)
        printed.append(current_model)

# Store some unprinted designs,
# and print each of them.
original = ['phone case', 'pendant', 'ring']
printed = []

print_models(original[:], printed)
print("\nOriginal:", original)
print("Printed:", printed)
```

Passing an arbitrary number of arguments

Sometimes you won't know how many arguments a function will need to accept. Python allows you to collect an arbitrary number of arguments into one parameter using the `*` operator. A parameter that accepts an arbitrary number of arguments must come last in the function definition.

The `**` operator allows a parameter to collect an arbitrary number of keyword arguments.

Collecting an arbitrary number of arguments

```
def make_pizza(size, *toppings):
    """Make a pizza."""
    print("\nMaking a " + size + " pizza.")
    print("Toppings:")
    for topping in toppings:
        print("- " + topping)

# Make three pizzas with different toppings.
make_pizza('small', 'pepperoni')
make_pizza('large', 'bacon bits', 'pineapple')
make_pizza('medium', 'mushrooms', 'peppers',
           'onions', 'extra cheese')
```

Collecting an arbitrary number of keyword arguments

```
def build_profile(first, last, **user_info):
    """Build a user's profile dictionary."""
    # Build a dict with the required keys.
    profile = {'first': first, 'last': last}

    # Add any other keys and values.
    for key, value in user_info.items():
        profile[key] = value

    return profile

# Create two users with different kinds
# of information.
user_0 = build_profile('albert', 'einstein',
                       location='princeton')
user_1 = build_profile('marie', 'curie',
                       location='paris', field='chemistry')

print(user_0)
print(user_1)
```

What's the best way to structure a function?

As you can see there are many ways to write and call a function. When you're starting out, aim for something that simply works. As you gain experience you'll develop an understanding of the more subtle advantages of different structures such as positional and keyword arguments, and the various approaches to importing functions. For now if your functions do what you need them to, you're doing well.

Modules

You can store your functions in a separate file called a module, and then import the functions you need into the file containing your main program. This allows for cleaner program files. (Make sure your module is stored in the same directory as your main program.)

Storing a function in a module

File: `pizza.py`

```
def make_pizza(size, *toppings):
    """Make a pizza."""
    print("\nMaking a " + size + " pizza.")
    print("Toppings:")
    for topping in toppings:
        print("- " + topping)
```

Importing an entire module

File: `making_pizzas.py`

Every function in the module is available in the program file.

```
import pizza
```

```
pizza.make_pizza('medium', 'pepperoni')
pizza.make_pizza('small', 'bacon', 'pineapple')
```

Importing a specific function

Only the imported functions are available in the program file.

```
from pizza import make_pizza
```

```
make_pizza('medium', 'pepperoni')
make_pizza('small', 'bacon', 'pineapple')
```

Giving a module an alias

```
import pizza as p
```

```
p.make_pizza('medium', 'pepperoni')
p.make_pizza('small', 'bacon', 'pineapple')
```

Giving a function an alias

```
from pizza import make_pizza as mp
```

```
mp('medium', 'pepperoni')
mp('small', 'bacon', 'pineapple')
```

Importing all functions from a module

Don't do this, but recognize it when you see it in others' code. It can result in naming conflicts, which can cause errors.

```
from pizza import *
```

```
make_pizza('medium', 'pepperoni')
make_pizza('small', 'bacon', 'pineapple')
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet - Classes

What are classes?

Classes are the foundation of object-oriented programming. Classes represent real-world things you want to model in your programs: for example dogs, cars, and robots. You use a class to make objects, which are specific instances of dogs, cars, and robots. A class defines the general behavior that a whole category of objects can have, and the information that can be associated with those objects.

Classes can inherit from each other – you can write a class that extends the functionality of an existing class. This allows you to code efficiently for a wide variety of situations.

Creating and using a class

Consider how we might model a car. What information would we associate with a car, and what behavior would it have? The information is stored in variables called attributes, and the behavior is represented by functions. Functions that are part of a class are called methods.

The Car class

```
class Car():
    """A simple attempt to model a car."""

    def __init__(self, make, model, year):
        """Initialize car attributes."""
        self.make = make
        self.model = model
        self.year = year

        # Fuel capacity and level in gallons.
        self.fuel_capacity = 15
        self.fuel_level = 0

    def fill_tank(self):
        """Fill gas tank to capacity."""
        self.fuel_level = self.fuel_capacity
        print("Fuel tank is full.")

    def drive(self):
        """Simulate driving."""
        print("The car is moving.")
```

Creating and using a class (cont.)

Creating an object from a class

```
my_car = Car('audi', 'a4', 2016)
```

Accessing attribute values

```
print(my_car.make)
print(my_car.model)
print(my_car.year)
```

Calling methods

```
my_car.fill_tank()
my_car.drive()
```

Creating multiple objects

```
my_car = Car('audi', 'a4', 2016)
my_old_car = Car('subaru', 'outback', 2013)
my_truck = Car('toyota', 'tacoma', 2010)
```

Modifying attributes

You can modify an attribute's value directly, or you can write methods that manage updating values more carefully.

Modifying an attribute directly

```
my_new_car = Car('audi', 'a4', 2016)
my_new_car.fuel_level = 5
```

Writing a method to update an attribute's value

```
def update_fuel_level(self, new_level):
    """Update the fuel level."""
    if new_level <= self.fuel_capacity:
        self.fuel_level = new_level
    else:
        print("The tank can't hold that much!")
```

Writing a method to increment an attribute's value

```
def add_fuel(self, amount):
    """Add fuel to the tank."""
    if (self.fuel_level + amount
        <= self.fuel_capacity):
        self.fuel_level += amount
        print("Added fuel.")
    else:
        print("The tank won't hold that much.")
```

Naming conventions

In Python class names are written in CamelCase and object names are written in lowercase with underscores. Modules that contain classes should still be named in lowercase with underscores.

Class inheritance

If the class you're writing is a specialized version of another class, you can use inheritance. When one class inherits from another, it automatically takes on all the attributes and methods of the parent class. The child class is free to introduce new attributes and methods, and override attributes and methods of the parent class.

To inherit from another class include the name of the parent class in parentheses when defining the new class.

The `__init__()` method for a child class

```
class ElectricCar(Car):
    """A simple model of an electric car."""

    def __init__(self, make, model, year):
        """Initialize an electric car."""
        super().__init__(make, model, year)

        # Attributes specific to electric cars.
        # Battery capacity in kWh.
        self.battery_size = 70
        # Charge level in %.
        self.charge_level = 0
```

Adding new methods to the child class

```
class ElectricCar(Car):
    --snip--
    def charge(self):
        """Fully charge the vehicle."""
        self.charge_level = 100
        print("The vehicle is fully charged.")
```

Using child methods and parent methods

```
my_ecar = ElectricCar('tesla', 'model s', 2016)

my_ecar.charge()
my_ecar.drive()
```

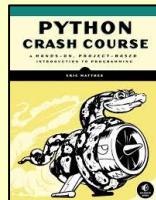
Finding your workflow

There are many ways to model real world objects and situations in code, and sometimes that variety can feel overwhelming. Pick an approach and try it – if your first attempt doesn't work, try a different approach.

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



Class inheritance (cont.)

Overriding parent methods

```
class ElectricCar(Car):
    --snip--
    def fill_tank(self):
        """Display an error message."""
        print("This car has no fuel tank!")
```

Instances as attributes

A class can have objects as attributes. This allows classes to work together to model complex situations.

A Battery class

```
class Battery():
    """A battery for an electric car."""

    def __init__(self, size=70):
        """Initialize battery attributes."""
        # Capacity in kWh, charge level in %.
        self.size = size
        self.charge_level = 0

    def get_range(self):
        """Return the battery's range."""
        if self.size == 70:
            return 240
        elif self.size == 85:
            return 270
```

Using an instance as an attribute

```
class ElectricCar(Car):
    --snip--

    def __init__(self, make, model, year):
        """Initialize an electric car."""
        super().__init__(make, model, year)

        # Attribute specific to electric cars.
        self.battery = Battery()

    def charge(self):
        """Fully charge the vehicle."""
        self.battery.charge_level = 100
        print("The vehicle is fully charged.")
```

Using the instance

```
my_ecar = ElectricCar('tesla', 'model x', 2016)

my_ecar.charge()
print(my_ecar.battery.get_range())
my_ecar.drive()
```

Importing classes

Class files can get long as you add detailed information and functionality. To help keep your program files uncluttered, you can store your classes in modules and import the classes you need into your main program.

Storing classes in a file

car.py

"""Represent gas and electric cars."""

```
class Car():
    """A simple attempt to model a car."""
    --snip--
```

```
class Battery():
    """A battery for an electric car."""
    --snip--
```

```
class ElectricCar(Car):
    """A simple model of an electric car."""
    --snip--
```

Importing individual classes from a module

```
from car import Car, ElectricCar

my_beetle = Car('volkswagen', 'beetle', 2016)
my_beetle.fill_tank()
my_beetle.drive()

my_tesla = ElectricCar('tesla', 'model s', 2016)
my_tesla.charge()
my_tesla.drive()
```

Importing an entire module

```
import car

my_beetle = car.Car(
    'volkswagen', 'beetle', 2016)
my_beetle.fill_tank()
my_beetle.drive()

my_tesla = car.ElectricCar(
    'tesla', 'model s', 2016)
my_tesla.charge()
my_tesla.drive()
```

Importing all classes from a module

(Don't do this, but recognize it when you see it.)

```
from car import *

my_beetle = Car('volkswagen', 'beetle', 2016)
```

Classes in Python 2.7

Classes should inherit from object

```
class ClassName(object):
```

The Car class in Python 2.7

```
class Car(object):
```

Child class `__init__()` method is different

```
class ChildClassName(ParentClass):
    def __init__(self):
        super(ClassName, self).__init__()
```

The ElectricCar class in Python 2.7

```
class ElectricCar(Car):
    def __init__(self, make, model, year):
        super(ElectricCar, self).__init__(
            make, model, year)
```

Storing objects in a list

A list can hold as many items as you want, so you can make a large number of objects from a class and store them in a list.

Here's an example showing how to make a fleet of rental cars, and make sure all the cars are ready to drive.

A fleet of rental cars

```
from car import Car, ElectricCar
```

```
# Make lists to hold a fleet of cars.
gas_fleet = []
electric_fleet = []
```

```
# Make 500 gas cars and 250 electric cars.
for _ in range(500):
    car = Car('ford', 'focus', 2016)
    gas_fleet.append(car)
for _ in range(250):
    ecar = ElectricCar('nissan', 'leaf', 2016)
    electric_fleet.append(ecar)
```

```
# Fill the gas cars, and charge electric cars.
for car in gas_fleet:
    car.fill_tank()
for ecar in electric_fleet:
    ecar.charge()
```

```
print("Gas cars:", len(gas_fleet))
print("Electric cars:", len(electric_fleet))
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Files and Exceptions

What are files? What are exceptions?

Your programs can read information in from files, and they can write data to files. Reading from files allows you to work with a wide variety of information; writing to files allows users to pick up where they left off the next time they run your program. You can write text to files, and you can store Python structures such as lists in data files.

Exceptions are special objects that help your programs respond to errors in appropriate ways. For example if your program tries to open a file that doesn't exist, you can use exceptions to display an informative error message instead of having the program crash.

Reading from a file

To read from a file your program needs to open the file and then read the contents of the file. You can read the entire contents of the file at once, or read the file line by line. The `with` statement makes sure the file is closed properly when the program has finished accessing the file.

Reading an entire file at once

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    contents = f_obj.read()

print(contents)
```

Reading line by line

Each line that's read from the file has a newline character at the end of the line, and the `print` function adds its own newline character. The `rstrip()` method gets rid of the extra blank lines this would result in when printing to the terminal.

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    for line in f_obj:
        print(line.rstrip())
```

Reading from a file (cont.)

Storing the lines in a list

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    lines = f_obj.readlines()

for line in lines:
    print(line.rstrip())
```

Writing to a file

Passing the 'w' argument to `open()` tells Python you want to write to the file. Be careful; this will erase the contents of the file if it already exists. Passing the 'a' argument tells Python you want to append to the end of an existing file.

Writing to an empty file

```
filename = 'programming.txt'

with open(filename, 'w') as f:
    f.write("I love programming!")
```

Writing multiple lines to an empty file

```
filename = 'programming.txt'

with open(filename, 'w') as f:
    f.write("I love programming!\n")
    f.write("I love creating new games.\n")
```

Appending to a file

```
filename = 'programming.txt'

with open(filename, 'a') as f:
    f.write("I also love working with data.\n")
    f.write("I love making apps as well.\n")
```

File paths

When Python runs the `open()` function, it looks for the file in the same directory where the program that's being executed is stored. You can open a file from a subfolder using a relative path. You can also use an absolute path to open any file on your system.

Opening a file from a subfolder

```
f_path = "text_files/alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()

for line in lines:
    print(line.rstrip())
```

File paths (cont.)

Opening a file using an absolute path

```
f_path = "/home/ehmatthes/books/alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()
```

Opening a file on Windows

Windows will sometimes interpret forward slashes incorrectly. If you run into this, use backslashes in your file paths.

```
f_path = "C:\Users\ehmatthes\books\alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()
```

The try-except block

When you think an error may occur, you can write a `try-except` block to handle the exception that might be raised. The `try` block tells Python to try running some code, and the `except` block tells Python what to do if the code results in a particular kind of error.

Handling the ZeroDivisionError exception

```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Handling the FileNotFoundError exception

```
f_name = 'siddhartha.txt'

try:
    with open(f_name) as f_obj:
        lines = f_obj.readlines()
except FileNotFoundError:
    msg = "Can't find file {}".format(f_name)
    print(msg)
```

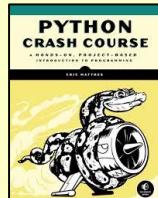
Knowing which exception to handle

It can be hard to know what kind of exception to handle when writing code. Try writing your code without a `try` block, and make it generate an error. The traceback will tell you what kind of exception your program needs to handle.

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



The else block

The `try` block should only contain code that may cause an error. Any code that depends on the `try` block running successfully should be placed in the `else` block.

Using an else block

```
print("Enter two numbers. I'll divide them.")

x = input("First number: ")
y = input("Second number: ")

try:
    result = int(x) / int(y)
except ZeroDivisionError:
    print("You can't divide by zero!")
else:
    print(result)
```

Preventing crashes from user input

Without the `except` block in the following example, the program would crash if the user tries to divide by zero. As written, it will handle the error gracefully and keep running.

```
"""A simple calculator for division only."""

print("Enter two numbers. I'll divide them.")
print("Enter 'q' to quit.")

while True:
    x = input("\nFirst number: ")
    if x == 'q':
        break
    y = input("Second number: ")
    if y == 'q':
        break

    try:
        result = int(x) / int(y)
    except ZeroDivisionError:
        print("You can't divide by zero!")
    else:
        print(result)
```

Deciding which errors to report

Well-written, properly tested code is not very prone to internal errors such as syntax or logical errors. But every time your program depends on something external such as user input or the existence of a file, there's a possibility of an exception being raised.

It's up to you how to communicate errors to your users. Sometimes users need to know if a file is missing; sometimes it's better to handle the error silently. A little experience will help you know how much to report.

Failing silently

Sometimes you want your program to just continue running when it encounters an error, without reporting the error to the user. Using the `pass` statement in an `else` block allows you to do this.

Using the pass statement in an else block

```
f_names = ['alice.txt', 'siddhartha.txt',
           'moby_dick.txt', 'little_women.txt']

for f_name in f_names:
    # Report the length of each file found.
    try:
        with open(f_name) as f_obj:
            lines = f_obj.readlines()
    except FileNotFoundError:
        # Just move on to the next file.
        pass
    else:
        num_lines = len(lines)
        msg = "{0} has {1} lines.".format(
            f_name, num_lines)
        print(msg)
```

Avoid bare except blocks

Exception-handling code should catch specific exceptions that you expect to happen during your program's execution. A bare `except` block will catch all exceptions, including keyboard interrupts and system exits you might need when forcing a program to close.

If you want to use a `try` block and you're not sure which exception to catch, use `Exception`. It will catch most exceptions, but still allow you to interrupt programs intentionally.

Don't use bare except blocks

```
try:
    # Do something
except:
    pass
```

Use Exception instead

```
try:
    # Do something
except Exception:
    pass
```

Printing the exception

```
try:
    # Do something
except Exception as e:
    print(e, type(e))
```

Storing data with json

The `json` module allows you to dump simple Python data structures into a file, and load the data from that file the next time the program runs. The JSON data format is not specific to Python, so you can share this kind of data with people who work in other languages as well.

Knowing how to manage exceptions is important when working with stored data. You'll usually want to make sure the data you're trying to load exists before working with it.

Using `json.dump()` to store data

```
"""Store some numbers."""

import json

numbers = [2, 3, 5, 7, 11, 13]

filename = 'numbers.json'
with open(filename, 'w') as f_obj:
    json.dump(numbers, f_obj)
```

Using `json.load()` to read data

```
"""Load some previously stored numbers."""

import json

filename = 'numbers.json'
with open(filename) as f_obj:
    numbers = json.load(f_obj)

print(numbers)
```

Making sure the stored data exists

```
import json

f_name = 'numbers.json'

try:
    with open(f_name) as f_obj:
        numbers = json.load(f_obj)
except FileNotFoundError:
    msg = "Can't find {0}.".format(f_name)
    print(msg)
else:
    print(numbers)
```

Practice with exceptions

Take a program you've already written that prompts for user input, and add some error-handling code to the program.

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — matplotlib

What is matplotlib?

Data visualization involves exploring data through visual representations. The matplotlib package helps you make visually appealing representations of the data you're working with. matplotlib is extremely flexible; these examples will help you get started with a few simple visualizations.

Installing matplotlib

matplotlib runs on all systems, but setup is slightly different depending on your OS. If the minimal instructions here don't work for you, see the more detailed instructions at <http://ehmatthes.github.io/pcc/>. You should also consider installing the Anaconda distribution of Python from <https://continuum.io/downloads/>, which includes matplotlib.

matplotlib on Linux

```
$ sudo apt-get install python3-matplotlib
```

matplotlib on OS X

Start a terminal session and enter `import matplotlib` to see if it's already installed on your system. If not, try this command:

```
$ pip install --user matplotlib
```

matplotlib on Windows

You first need to install Visual Studio, which you can do from <https://dev.windows.com/>. The Community edition is free. Then go to <https://pypi.python.org/pypi/matplotlib/> or <http://www.lfd.uci.edu/~gohlke/pythonlibs/#matplotlib> and download an appropriate installer file.

Line graphs and scatter plots

Making a line graph

```
import matplotlib.pyplot as plt

x_values = [0, 1, 2, 3, 4, 5]
squares = [0, 1, 4, 9, 16, 25]
plt.plot(x_values, squares)
plt.show()
```

Line graphs and scatter plots (cont.)

Making a scatter plot

The `scatter()` function takes a list of `x` values and a list of `y` values, and a variety of optional arguments. The `s=10` argument controls the size of each point.

```
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]

plt.scatter(x_values, squares, s=10)
plt.show()
```

Customizing plots

Plots can be customized in a wide variety of ways. Just about any element of a plot can be customized.

Adding titles and labels, and scaling axes

```
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]
plt.scatter(x_values, squares, s=10)

plt.title("Square Numbers", fontsize=24)
plt.xlabel("Value", fontsize=18)
plt.ylabel("Square of Value", fontsize=18)
plt.tick_params(axis='both', which='major',
                labelsize=14)
plt.axis([0, 1100, 0, 1100000])

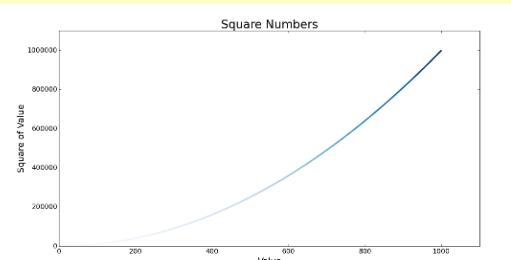
plt.show()
```

Using a colormap

A colormap varies the point colors from one shade to another, based on a certain value for each point. The value used to determine the color of each point is passed to the `c` argument, and the `cmap` argument specifies which colormap to use.

The `edgecolor='none'` argument removes the black outline from each point.

```
plt.scatter(x_values, squares, c=squares,
            cmap=plt.cm.Blues, edgecolor='none',
            s=10)
```



Customizing plots (cont.)

Emphasizing points

You can plot as much data as you want on one plot. Here we re-plot the first and last points larger to emphasize them.

```
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]
plt.scatter(x_values, squares, c=squares,
            cmap=plt.cm.Blues, edgecolor='none',
            s=10)

plt.scatter(x_values[0], squares[0], c='green',
            edgecolor='none', s=100)
plt.scatter(x_values[-1], squares[-1], c='red',
            edgecolor='none', s=100)

plt.title("Square Numbers", fontsize=24)
--snip--
```

Removing axes

You can customize or remove axes entirely. Here's how to access each axis, and hide it.

```
plt.axes().get_xaxis().set_visible(False)
plt.axes().get_yaxis().set_visible(False)
```

Setting a custom figure size

You can make your plot as big or small as you want. Before plotting your data, add the following code. The `dpi` argument is optional; if you don't know your system's resolution you can omit the argument and adjust the `figsize` argument accordingly.

```
plt.figure(dpi=128, figsize=(10, 6))
```

Saving a plot

The matplotlib viewer has an interactive save button, but you can also save your visualizations programmatically. To do so, replace `plt.show()` with `plt.savefig()`. The `bbox_inches='tight'` argument trims extra whitespace from the plot.

```
plt.savefig('squares.png', bbox_inches='tight')
```

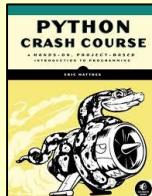
Online resources

The matplotlib gallery and documentation are at <http://matplotlib.org/>. Be sure to visit the examples, gallery, and pyplot links.

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



Multiple plots

You can make as many plots as you want on one figure. When you make multiple plots, you can emphasize relationships in the data. For example you can fill the space between two sets of data.

Plotting two sets of data

Here we use `plt.scatter()` twice to plot square numbers and cubes on the same figure.

```
import matplotlib.pyplot as plt

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

plt.scatter(x_values, squares, c='blue',
            edgecolor='none', s=20)
plt.scatter(x_values, cubes, c='red',
            edgecolor='none', s=20)

plt.axis([0, 11, 0, 1100])
plt.show()
```

Filling the space between data sets

The `fill_between()` method fills the space between two data sets. It takes a series of x-values and two series of y-values. It also takes a facecolor to use for the fill, and an optional alpha argument that controls the color's transparency.

```
plt.fill_between(x_values, cubes, squares,
                 facecolor='blue', alpha=0.25)
```

Working with dates and times

Many interesting data sets have a date or time as the x-value. Python's `datetime` module helps you work with this kind of data.

Generating the current date

The `datetime.now()` function returns a `datetime` object representing the current date and time.

```
from datetime import datetime as dt

today = dt.now()
date_string = dt.strftime(today, '%m/%d/%Y')
print(date_string)
```

Generating a specific date

You can also generate a `datetime` object for any date and time you want. The positional order of arguments is year, month, and day. The hour, minute, second, and microsecond arguments are optional.

```
from datetime import datetime as dt

new_years = dt(2017, 1, 1)
fall_equinox = dt(year=2016, month=9, day=22)
```

Working with dates and times (cont.)

Datetime formatting arguments

The `strftime()` function generates a formatted string from a `datetime` object, and the `strptime()` function generates a `datetime` object from a string. The following codes let you work with dates exactly as you need to.

%A	Weekday name, such as Monday
%B	Month name, such as January
%m	Month, as a number (01 to 12)
%d	Day of the month, as a number (01 to 31)
%Y	Four-digit year, such as 2016
%y	Two-digit year, such as 16
%H	Hour, in 24-hour format (00 to 23)
%I	Hour, in 12-hour format (01 to 12)
%p	AM or PM
%M	Minutes (00 to 59)
%S	Seconds (00 to 61)

Converting a string to a datetime object

```
new_years = dt.strptime('1/1/2017', '%m/%d/%Y')
```

Converting a datetime object to a string

```
ny_string = dt.strftime(new_years, '%B %d, %Y')
print(ny_string)
```

Plotting high temperatures

The following code creates a list of dates and a corresponding list of high temperatures. It then plots the high temperatures, with the date labels displayed in a specific format.

```
from datetime import datetime as dt

import matplotlib.pyplot as plt
from matplotlib import dates as mdates

dates = [
    dt(2016, 6, 21), dt(2016, 6, 22),
    dt(2016, 6, 23), dt(2016, 6, 24),
]

highs = [57, 68, 64, 59]

fig = plt.figure(dpi=128, figsize=(10,6))
plt.plot(dates, highs, c='red')
plt.title("Daily High Temps", fontsize=24)
plt.ylabel("Temp (F)", fontsize=16)

x_axis = plt.axes().get_xaxis()
x_axis.set_major_formatter(
    mdates.DateFormatter('%B %d %Y')
)
fig.autofmt_xdate()

plt.show()
```

Multiple plots in one figure

You can include as many individual graphs in one figure as you want. This is useful, for example, when comparing related datasets.

Sharing an x-axis

The following code plots a set of squares and a set of cubes on two separate graphs that share a common x-axis.

The `plt.subplots()` function returns a figure object and a tuple of axes. Each set of axes corresponds to a separate plot in the figure. The first two arguments control the number of rows and columns generated in the figure.

```
import matplotlib.pyplot as plt

x_vals = list(range(11))
squares = [x**2 for x in x_vals]
cubes = [x**3 for x in x_vals]

fig, axarr = plt.subplots(2, 1, sharex=True)

axarr[0].scatter(x_vals, squares)
axarr[0].set_title('Squares')

axarr[1].scatter(x_vals, cubes, c='red')
axarr[1].set_title('Cubes')

plt.show()
```

Sharing a y-axis

To share a y-axis, we use the `sharey=True` argument.

```
import matplotlib.pyplot as plt

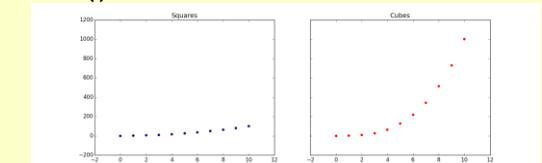
x_vals = list(range(11))
squares = [x**2 for x in x_vals]
cubes = [x**3 for x in x_vals]

fig, axarr = plt.subplots(1, 2, sharey=True)

axarr[0].scatter(x_vals, squares)
axarr[0].set_title('Squares')

axarr[1].scatter(x_vals, cubes, c='red')
axarr[1].set_title('Cubes')

plt.show()
```



More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Pygal

What is Pygal?

Data visualization involves exploring data through visual representations. Pygal helps you make visually appealing representations of the data you're working with. Pygal is particularly well suited for visualizations that will be presented online, because it supports interactive elements.

Installing Pygal

Pygal can be installed using pip.

Pygal on Linux and OS X

```
$ pip install --user pygal
```

Pygal on Windows

```
> python -m pip install --user pygal
```

Line graphs, scatter plots, and bar graphs

To make a plot with Pygal, you specify the kind of plot and then add the data.

Making a line graph

To view the output, open the file squares.svg in a browser.

```
import pygal

x_values = [0, 1, 2, 3, 4, 5]
squares = [0, 1, 4, 9, 16, 25]

chart = pygal.Line()
chart.force_uri_protocol = 'http'
chart.add('x^2', squares)
chart.render_to_file('squares.svg')
```

Adding labels and a title

```
--snip--
chart = pygal.Line()
chart.force_uri_protocol = 'http'
chart.title = "Squares"
chart.x_labels = x_values
chart.x_title = "Value"
chart.y_title = "Square of Value"
chart.add('x^2', squares)
chart.render_to_file('squares.svg')
```

Line graphs, scatter plots, and bar graphs (cont.)

Making a scatter plot

The data for a scatter plot needs to be a list containing tuples of the form (x, y) . The `stroke=False` argument tells Pygal to make an XY chart with no line connecting the points.

```
import pygal

squares = [
    (0, 0), (1, 1), (2, 4), (3, 9),
    (4, 16), (5, 25),
]

chart = pygal.XY(stroke=False)
chart.force_uri_protocol = 'http'
chart.add('x^2', squares)
chart.render_to_file('squares.svg')
```

Using a list comprehension for a scatter plot

A list comprehension can be used to efficiently make a dataset for a scatter plot.

```
squares = [(x, x**2) for x in range(1000)]
```

Making a bar graph

A bar graph requires a list of values for the bar sizes. To label the bars, pass a list of the same length to `x_labels`.

```
import pygal

outcomes = [1, 2, 3, 4, 5, 6]
frequencies = [18, 16, 18, 17, 18, 13]

chart = pygal.Bar()
chart.force_uri_protocol = 'http'
chart.x_labels = outcomes
chart.add('D6', frequencies)
chart.render_to_file('rolling_dice.svg')
```

Making a bar graph from a dictionary

Since each bar needs a label and a value, a dictionary is a great way to store the data for a bar graph. The keys are used as the labels along the x-axis, and the values are used to determine the height of each bar.

```
import pygal

results = {
    1:18, 2:16, 3:18,
    4:17, 5:18, 6:13,
}

chart = pygal.Bar()
chart.force_uri_protocol = 'http'
chart.x_labels = results.keys()
chart.add('D6', results.values())
chart.render_to_file('rolling_dice.svg')
```

Multiple plots

You can add as much data as you want when making a visualization.

Plotting squares and cubes

```
import pygal

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

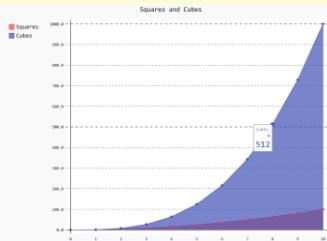
chart = pygal.Line()
chart.force_uri_protocol = 'http'
chart.title = "Squares and Cubes"
chart.x_labels = x_values

chart.add('Squares', squares)
chart.add('Cubes', cubes)
chart.render_to_file('squares_cubes.svg')
```

Filling the area under a data series

Pygal allows you to fill the area under or over each series of data. The default is to fill from the x-axis up, but you can fill from any horizontal line using the `zero` argument.

```
chart = pygal.Line(fill=True, zero=0)
```



Online resources

The documentation for Pygal is available at <http://www.pygal.org/>.

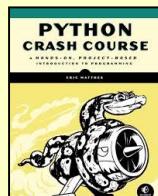
Enabling interactive features

If you're viewing svg output in a browser, Pygal needs to render the output file in a specific way. The `force_uri_protocol` attribute for `chart` objects needs to be set to 'http'.

Python Crash Course

[Covers Python 3 and Python 2](#)

nostarchpress.com/pythoncrashcourse



Styling plots

Pygal lets you customize many elements of a plot. There are some excellent default themes, and many options for styling individual plot elements.

Using built-in styles

To use built-in styles, import the style and make an instance of the style class. Then pass the style object with the style argument when you make the chart object.

```
import pygal
from pygal.style import LightGreenStyle

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

chart_style = LightGreenStyle()
chart = pygal.Line(style=chart_style)
chart.force_uri_protocol = 'http'
chart.title = "Squares and Cubes"
chart.x_labels = x_values

chart.add('Squares', squares)
chart.add('Cubes', cubes)
chart.render_to_file('squares_cubes.svg')
```

Parametric built-in styles

Some built-in styles accept a custom color, then generate a theme based on that color.

```
from pygal.style import LightenStyle

--snip--
chart_style = LightenStyle('#336688')
chart = pygal.Line(style=chart_style)
--snip--
```

Customizing individual style properties

Style objects have a number of properties you can set individually.

```
chart_style = LightenStyle('#336688')
chart_style.plot_background = '#CCCCCC'
chart_style.major_label_font_size = 20
chart_style.label_font_size = 16
--snip--
```

Custom style class

You can start with a bare style class, and then set only the properties you care about.

```
chart_style = Style()
chart_style.colors = [
    '#CCCCCC', '#AAAAAA', '#888888']
chart_style.plot_background = '#EEEEEE'

chart = pygal.Line(style=chart_style)
--snip--
```

Styling plots (cont.)

Configuration settings

Some settings are controlled by a Config object.

```
my_config = pygal.Config()
my_config.show_y_guides = False
my_config.width = 1000
my_config.dots_size = 5

chart = pygal.Line(config=my_config)
--snip--
```

Styling series

You can give each series on a chart different style settings.

```
chart.add('Squares', squares, dots_size=2)
chart.add('Cubes', cubes, dots_size=3)
```

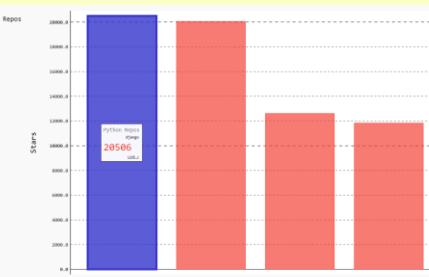
Styling individual data points

You can style individual data points as well. To do so, write a dictionary for each data point you want to customize. A 'value' key is required, and other properties are optional.

```
import pygal

repos = [
    {
        'value': 20506,
        'color': '#3333CC',
        'xlink': 'http://djangoproject.com/',
    },
    20054,
    12607,
    11827,
]
```

```
chart = pygal.Bar()
chart.force_uri_protocol = 'http'
chart.x_labels = [
    'django', 'requests', 'scikit-learn',
    'tornado',
]
chart.y_title = 'Stars'
chart.add('Python Repos', repos)
chart.render_to_file('python_repos.svg')
```



Plotting global datasets

Pygal can generate world maps, and you can add any data you want to these maps. Data is indicated by coloring, by labels, and by tooltips that show data when users hover over each country on the map.

Installing the world map module

The world map module is not included by default in Pygal 2.0. It can be installed with pip:

```
$ pip install --user pygal_maps_world
```

Making a world map

The following code makes a simple world map showing the countries of North America.

```
from pygal.maps.world import World
```

```
wm = World()
wm.force_uri_protocol = 'http'
wm.title = 'North America'
wm.add('North America', ['ca', 'mx', 'us'])

wm.render_to_file('north_america.svg')
```

Showing all the country codes

In order to make maps, you need to know Pygal's country codes. The following example will print an alphabetical list of each country and its code.

```
from pygal.maps.world import COUNTRIES

for code in sorted(COUNTRIES.keys()):
    print(code, COUNTRIES[code])
```

Plotting numerical data on a world map

To plot numerical data on a map, pass a dictionary to add() instead of a list.

```
from pygal.maps.world import World

populations = {
    'ca': 34126000,
    'us': 309349000,
    'mx': 113423000,
}
```

```
wm = World()
wm.force_uri_protocol = 'http'
wm.title = 'Population of North America'
wm.add('North America', populations)
```

```
wm.render_to_file('na_populations.svg')
```

[More cheat sheets available at
ehmatthes.github.io/pcc/](https://ehmatthes.github.io/pcc/)



Real Python Pocket Reference

Visit realpython.com to turbocharge your Python learning with in-depth tutorials, real-world examples, and expert guidance.

Getting Started

Follow these guides to kickstart your Python journey:

- [realpython.com/what-can-i-do-with-python](#)
- [realpython.com/installing-python](#)
- [realpython.com/python-first-steps](#)

Start the Interactive Shell

```
$ python
```

Quit the Interactive Shell

```
>>> exit()
```

Run a Script

```
$ python my_script.py
```

Run a Script in Interactive Mode

```
$ python -i my_script.py
```

Learn More on [realpython.com/search](#):

[interpreter](#) · [run a script](#) · [command line](#)

Comments

- Always add a space after the #
- Use comments to explain "why" of your code

Write Comments

```
# This is a comment
# print("This code will not run.")
print("This will run.") # Comments are ignored by Python
```

Learn More on [realpython.com/search](#):

[comment](#) · [documentation](#)

Data Types

- Python is dynamically typed
- Use `None` to represent missing or optional values
- Use `type()` to check object type
- Check for a specific type with `isinstance()`
- `issubclass()` checks if a class is a subclass

Type Investigation

```
type(42)           # <class 'int'>
type(3.14)         # <class 'float'>
type("Hello")      # <class 'str'>
type(True)          # <class 'bool'>
type(None)          # <class 'NoneType'>

isinstance(3.14, float) # True
issubclass(int, object) # True - everything inherits from object
```

Type Conversion

```
int("42")          # 42
float("3.14")      # 3.14
str(42)            # "42"
bool(1)             # True
list("abc")         # ["a", "b", "c"]
```

Learn More on [realpython.com/search](#):

[data types](#) · [type checking](#) · [isinstance](#) · [issubclass](#)

Variables & Assignment

- Variables are created when first assigned
- Use descriptive variable names
- Follow `snake_case` convention

Basic Assignment

```
name = "Leo"          # String
age = 7               # Integer
height = 5.6          # Float
is_cat = True          # Boolean
flaws = None           # None type
```

Parallel & Chained Assignments

```
x, y = 10, 20        # Assign multiple values
a = b = c = 0          # Give same value to multiple variables
```

Augmented Assignments

```
counter += 1
numbers += [4, 5]
permissions |= write
```

Learn More on [realpython.com/search](#):

[variables](#) · [assignment operator](#) · [walrus operator](#)

Strings

- It's recommended to use double-quotes for strings
- Use `\n` to create a line break in a string
- To write a backslash in a normal string, write `\\"\\`

Creating Strings

```
single = 'Hello'
double = "World"
multi = """Multiple
line string"""
```

String Operations

```
greeting = "me" + "ow!" # "meow!"
repeat = "Meow!" * 3     # "Meow!Meow!Meow!"
length = len("Python")   # 6
```

String Methods

```
"a".upper()           # "A"
"A".lower()           # "a"
" a ".strip()          # "a"
"abc".replace("bc", "ha") # "aha"
"a b".split()          # ["a", "b"]
"-".join(["a", "b"])    # "a-b"
```

String Indexing & Slicing

```
text = "Python"
text[0]               # "P" (first)
text[-1]              # "n" (last)
text[1:4]              # "yth" (slice)
text[:3]               # "Pyt" (from start)
text[3:]               # "hon" (to end)
text[::-2]              # "Pto" (every 2nd)
text[::-1]              # "nohtyP" (reverse)
```

String Formatting

```
# f-strings
name = "Aubrey"
age = 2
f"Hello, {name}!"           # "Hello, Aubrey!"
f"{name} is {age} years old" # "Aubrey is 2 years old"
f"Debug: {age=}"            # "Debug: age=2"

# Format method
template = "Hello, {name}! You're {age}."
template.format(name="Aubrey", age=2) # "Hello, Aubrey! You're 2."
```

Raw Strings

```
# Normal string with an escaped tab
"This is:\tCool."       # "This is: Cool."
# Raw string with escape sequences
r"This is:\tCool."      # "This is:\tCool."
```

Learn More on [realpython.com/search](#):

[strings](#) · [string methods](#) · [slice notation](#) · [raw strings](#)

Numbers & Math

Arithmetic Operators

```
10 + 3    # 13
10 - 3    # 7
10 * 3    # 30
10 / 3    # 3.3333333333333335
10 // 3   # 3
10 % 3    # 1
2 ** 3    # 8
```

Useful Functions

```
abs(-5)      # 5
round(3.7)    # 4
round(3.14159, 2) # 3.14
min(3, 1, 2)  # 1
max(3, 1, 2)  # 3
sum([1, 2, 3]) # 6
```

Learn More on [realpython.com/search](#):

[math](#) · [operators](#) · [built in functions](#)

Conditionals

- Python uses indentation for code blocks
- Use 4 spaces per indentation level

If-Elif-Else

```
if age < 13:
    category = "child"
elif age < 20:
    category = "teenager"
else:
    category = "adult"
```

Comparison Operators

```
x == y    # Equal to
x != y    # Not equal to
x < y     # Less than
x <= y    # Less than or equal
x > y     # Greater than
x >= y    # Greater than or equal
```

Logical Operators

```
if age >= 18 and has_car:
    print("Roadtrip!")

if is_weekend or is_holiday:
    print("No work today.")

if not is_raining:
    print("You can go outside.")
```

Learn More on [realpython.com/search](#):

[conditional statements](#) · [operators](#) · [truthy falsy](#)

Loops

- `range(5)` generates 0 through 4
- Use `enumerate()` to get index and value
- `break` exits the loop, `continue` skips to next
- Be careful with `while` to not create an infinite loop

For Loops

```
# Loop through range
for i in range(5):      # 0, 1, 2, 3, 4
    print(i)

# Loop through collection
fruits = ["apple", "banana"]
for fruit in fruits:
    print(fruit)

# With enumerate for index
for i, fruit in enumerate(fruits):
    print(f"{i}: {fruit}")
```

While Loops

```
while True:
    user_input = input("Enter 'quit' to exit: ")
    if user_input == "quit":
        break
    print(f"You entered: {user_input}")
```

Loop Control

```
for i in range(10):
    if i == 3:
        continue # Skip this iteration
    if i == 7:
        break   # Exit loop
    print(i)
```

Learn More on realpython.com/search:

for loop · while loop · enumerate · control flow

Functions

- Define functions with `def`
- Always use `()` to call a function
- Add `return` to send values back
- Create anonymous functions with the `lambda` keyword

Defining Functions

```
def greet():
    return "Hello!"

def greet_person(name):
    return f"Hello, {name}!"

def add(x, y=10):    # Default parameter
    return x + y
```

Calling Functions

```
greet()           # "Hello!"
greet_person("Bartosz") # "Hello, Bartosz"
add(5, 3)         # 8
add(7)           # 17
```

Return Values

```
def get_min_max(numbers):
    return min(numbers), max(numbers)

minimum, maximum = get_min_max([1, 5, 3])
```

Useful Built-in Functions

```
callable() # Checks if an object can be called as a function
dir()     # Lists attributes and methods
globals() # Get a dictionary of the current global symbol table
hash()    # Get the hash value
id()     # Get the unique identifier
locals() # Get a dictionary of the current local symbol table
repr()   # Get a string representation for debugging
```

Lambda Functions

```
square = lambda x: x**2
result = square(5) # 25

# With map and filter
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, numbers))
evens = list(filter(lambda x: x % 2 == 0, numbers))
```

Learn More on realpython.com/search:

define functions · return multiple values · lambda

Classes

- Classes are blueprints for objects
- You can create multiple instances of one class
- You commonly use classes to encapsulate data
- Inside a class, you provide methods for interacting with the data
- `__init__()` is the constructor method
- `self` refers to the instance

Defining Classes

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        return f"{self.name} says Woof!"

# Create instance
my_dog = Dog("Frieda", 3)
print(my_dog.bark()) # Frieda says Woof!
```

Class Attributes & Methods

```
class Cat:
    species = "Felis catus" # Class attribute

    def __init__(self, name):
        self.name = name      # Instance attribute

    def meow(self):
        return f"{self.name} says Meow!"

@classmethod
def create_kitten(cls, name):
    return cls(f"Baby {name}")
```

Inheritance

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return f"{self.name} barks!"
```

Learn More on realpython.com/search:

object oriented programming · classes

Exceptions

- When Python runs and encounters an error, it creates an exception
- Use specific exception types when possible
- `else` runs if no exception occurred
- `finally` always runs, even after errors

Try-Except

```
try:
    number = int(input("Enter a number: "))
    result = 10 / number
except ValueError:
    print("That's not a valid number!")
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    print(f"Result: {result}")
finally:
    print("Calculation attempted")
```

Common Exceptions

ValueError	# Invalid value
TypeError	# Wrong type
IndexError	# List index out of range
KeyError	# Dict key not found
FileNotFoundException	# File doesn't exist

Raising Exceptions

```
def validate_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative")
    return age
```

Learn More on realpython.com/search:
exceptions · errors · debugging

Collections

- A collection is any container data structure that stores multiple items
- If an object is a collection, then you can loop through it
- Strings are collections, too
- Use `len()` to get the size of a collection
- You can check if an item is in a collection with the `in` keyword
- Some collections may look similar, but each data structure solves specific needs

Lists

```
# Creating lists
empty = []
nums = [5]
mixed = [1, "two", 3.0, True]

# List methods
nums.append("x")          # Add to end
nums.insert(0, "y")         # Insert at index 0
nums.extend(["z", 5])       # Extend with iterable
nums.remove("x")           # Remove first "x"
last = nums.pop()          # Pop returns last element
```

```
# List indexing and checks
fruits = ["banana", "apple", "orange"]
fruits[0]                  # "banana"
fruits[-1]                 # "orange"
"apple" in fruits          # True
len(fruits)                # 3
```

Tuples

```
# Creating tuples
point = (3, 4)
single = (1,)    # Note the comma!
empty = ()
```

```
# Basic tuple unpacking
point = (3, 4)
x, y = point
x                      # 3
y                      # 4
```

```
# Extended unpacking
first, *rest = (1, 2, 3, 4)
first                  # 1
rest                   # [2, 3, 4]
```

Sets

```
# Creating Sets
a = {1, 2, 3}
b = set([3, 4, 4, 5])

# Set Operations
a | b      # {1, 2, 3, 4, 5}
a & b      # {3}
a - b      # {1, 2}
a ^ b      # {1, 2, 4, 5}
```

Dictionaries

```
# Creating Dictionaries
empty = {}
pet = {"name": "Leo", "age": 42}

# Dictionary Operations
pet["sound"] = "Purr!" # Add key and value
pet["age"] = 7 # Update value
age = pet.get("age", 0) # Get with default
del pet["sound"] # Delete key
pet.pop("age") # Remove and return

# Dictionary Methods
pet = {"name": "Frieda", "sound": "Bark!"}
pet.keys() # dict_keys(['name', 'sound'])
pet.values() # dict_values(['Frieda', 'Bark!'])
pet.items() # dict_items([('name', 'Frieda'), ('sound', 'Bark!')])
```

Learn More on realpython.com/search:

list · tuple · set · dictionary · indexing · unpacking

Comprehensions

- You can think of comprehensions as condensed `for` loops
- Comprehensions are faster than equivalent loops

List Comprehensions

```
# Basic
squares = [x**2 for x in range(10)]

# With condition
evens = [x for x in range(20) if x % 2 == 0]

# Nested
matrix = [[i*j for j in range(3)] for i in range(3)]
```

Other Comprehensions

```
# Dictionary comprehension
word_lengths = {word: len(word) for word in ["hello", "world"]}

# Set comprehension
unique_lengths = {len(word) for word in ["who", "what", "why"]}

# Generator expression
sum_squares = sum(x**2 for x in range(1000))
```

Learn More on realpython.com/search:

comprehensions · data structures · generators

File I/O

File Operations

```
# Read an entire file
with open("file.txt", mode="r", encoding="utf-8") as file:
    content = file.read()

# Read a file line by line
with open("file.txt", mode="r", encoding="utf-8") as file:
    for line in file:
        print(line.strip())

# Write a file
with open("output.txt", mode="w", encoding="utf-8") as file:
    file.write("Hello, World!\n")

# Append to a File
with open("log.txt", mode="a", encoding="utf-8") as file:
    file.write("New log entry\n")
```

Learn More on realpython.com/search:

files · context manager · pathlib

Imports & Modules

- Prefer explicit imports over `import *`
- Use aliases for long module names
- Group imports: standard library, third-party libraries, user-defined modules

Import Styles

```
# Import entire module
import math
result = math.sqrt(16)

# Import specific function
from math import sqrt
result = sqrt(16)

# Import with alias
import numpy as np
array = np.array([1, 2, 3])

# Import all (not recommended)
from math import *
```

Package Imports

```
# Import from package
import package.module
from package import module
from package.subpackage import module

# Import specific items
from package.module import function, Class
from package.module import name as alias
```

Learn More on realpython.com/search:

import · modules · packages

Virtual Environments

- Virtual Environments are often called “venv”
- Use venvs to isolate project packages from the system-wide Python packages

Create Virtual Environment

```
$ python -m venv .venv
```

Activate Virtual Environment (Windows)

```
PS> .venv\Scripts\activate
```

Activate Virtual Environment (Linux & macOS)

```
$ source .venv/bin/activate
```

Deactivate Virtual Environment

```
(.venv) $ deactivate
```

Learn More on realpython.com/search:

virtual environment · venv

Packages

- The official third-party package repository is the Python Package Index (PyPI)

Install Packages

```
$ python -m pip install requests
```

Save Requirements & Install from File

```
$ python -m pip freeze > requirements.txt
$ python -m pip install -r requirements.txt
```

Related Tutorials

- Installing Python Packages
- Requirements Files in Python Projects

Miscellaneous

Truthy

-42

3.14

"John"

[1, 2, 3]

("apple", "banana")

{"key": None}

Falsy

0

0.0

""

[]

()

{}

None

Pythonic Constructs

```
# Swap variables
a, b = b, a
```

```
# Flatten a list of lists
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat = [item for sublist in matrix for item in sublist]
```

```
# Remove duplicates
unique_unordered = list(set(my_list))
```

```
# Remove duplicates, preserve order
unique = list(dict.fromkeys(my_list))
```

```
# Count occurrences
from collections import Counter
counts = Counter(my_list)
```

Learn More on realpython.com/search:
counter · tricks

Do you want to go deeper on any topic in the Python curriculum?

At Real Python you can immerse yourself in any topic. Level up your skills effectively with curated resources like:

- Learning paths
- Video courses
- Written tutorials
- Interactive quizzes
- Podcast interviews
- Reference articles

Continue your learning journey and become a Python expert at realpython.com/start-here💡🧠

Python For Data Science Cheat Sheet

Python Basics

Learn More Python for Data Science [Interactively](#) at www.datacamp.com



Variables and Data Types

Variable Assignment

```
>>> x=5  
>>> x  
5
```

Calculations With Variables

>>> x+2 7	Sum of two variables
>>> x-2 3	Subtraction of two variables
>>> x*2 10	Multiplication of two variables
>>> x**2 25	Exponentiation of a variable
>>> x%2 1	Remainder of a variable
>>> x/float(2) 2.5	Division of a variable

Types and Type Conversion

str()	'5', '3.45', 'True'	Variables to strings
int()	5, 3, 1	Variables to integers
float()	5.0, 1.0	Variables to floats
bool()	True, True, True	Variables to booleans

Asking For Help

```
>>> help(str)
```

Strings

```
>>> my_string = 'thisStringIsAwesome'  
>>> my_string  
'thisStringIsAwesome'
```

String Operations

```
>>> my_string * 2  
'thisStringIsAwesomethisStringIsAwesome'  
>>> my_string + 'Innit'  
'thisStringIsAwesomeInnit'  
>>> 'm' in my_string  
True
```

Lists

```
>>> a = 'is'  
>>> b = 'nice'  
>>> my_list = ['my', 'list', a, b]  
>>> my_list2 = [[4,5,6,7], [3,4,5,6]]
```

Selecting List Elements

Index starts at 0

Subset

```
>>> my_list[1]  
>>> my_list[-3]
```

Slice

```
>>> my_list[1:3]  
>>> my_list[1:]
```

```
>>> my_list[:3]  
>>> my_list[:]
```

Subset Lists of Lists

```
>>> my_list2[1][0]  
>>> my_list2[1][:2]
```

Select item at index 1
Select 3rd last item

Select items at index 1 and 2
Select items after index 0

Select items before index 3

Copy my_list

my_list[list][itemOfList]

List Operations

```
>>> my_list + my_list  
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']  
>>> my_list * 2  
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']  
>>> my_list2 > 4  
True
```

List Methods

```
>>> my_list.index('a')  
>>> my_list.count('a')  
>>> my_list.append('!')  
>>> my_list.remove('!')  
>>> del(my_list[0:1])  
>>> my_list.reverse()  
>>> my_list.extend('!')  
>>> my_list.pop(-1)  
>>> my_list.insert(0, '!')  
>>> my_list.sort()
```

Get the index of an item
Count an item
Append an item at a time
Remove an item
Remove an item
Reverse the list
Append an item
Remove an item
Insert an item
Sort the list

String Operations

Index starts at 0

```
>>> my_string[3]  
>>> my_string[4:9]
```

String Methods

```
>>> my_string.upper()  
>>> my_string.lower()  
>>> my_string.count('w')  
>>> my_string.replace('e', 'i')  
>>> my_string.strip()
```

String to uppercase
String to lowercase
Count String elements
Replace String elements
Strip whitespaces

Also see NumPy Arrays

Libraries

Import libraries

```
>>> import numpy  
>>> import numpy as np  
Selective import  
>>> from math import pi
```

pandas Data analysis

Machine learning

NumPy Scientific computing

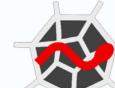
matplotlib 2D plotting

Install Python



ANACONDA®

Leading open data science platform
powered by Python



spyder
Free IDE that is included
with Anaconda



Create and share
documents with live code,
visualizations, text, ...

Numpy Arrays

Also see Lists

```
>>> my_list = [1, 2, 3, 4]  
>>> my_array = np.array(my_list)  
>>> my_2darray = np.array([[1,2,3],[4,5,6]])
```

Selecting Numpy Array Elements

Index starts at 0

Subset

```
>>> my_array[1]  
2
```

Select item at index 1

Slice

```
>>> my_array[0:2]  
array([1, 2])
```

Select items at index 0 and 1

Subset 2D Numpy arrays

```
>>> my_2darray[:,0]  
array([1, 4])
```

my_2darray[rows, columns]

Numpy Array Operations

```
>>> my_array > 3  
array([False, False, False, True], dtype=bool)  
>>> my_array * 2  
array([2, 4, 6, 8])  
>>> my_array + np.array([5, 6, 7, 8])  
array([6, 8, 10, 12])
```

Numpy Array Functions

```
>>> my_array.shape  
>>> np.append(other_array)  
>>> np.insert(my_array, 1, 5)  
>>> np.delete(my_array, [1])  
>>> np.mean(my_array)  
>>> np.median(my_array)  
>>> my_array.corrcoef()  
>>> np.std(my_array)
```

Get the dimensions of the array
Append items to an array
Insert items in an array
Delete items in an array
Mean of the array
Median of the array
Correlation coefficient
Standard deviation



Python For Data Science Cheat Sheet

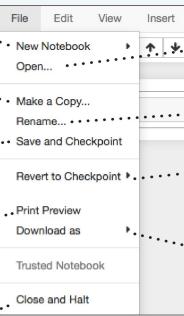
Jupyter Notebook

Learn More Python for Data Science Interactively at www.DataCamp.com



Saving/Loading Notebooks

Create new notebook



Make a copy of the current notebook

Open an existing notebook

Rename notebook

Revert notebook to a previous checkpoint

Download notebook as
- IPython notebook
- Python
- HTML
- Markdown
- reST
- LaTeX
- PDF

Save current notebook and record checkpoint

Preview of the printed notebook

Close notebook & stop running any scripts

Writing Code And Text

Code and text are encapsulated by 3 basic cell types: markdown cells, code cells, and raw NBConvert cells.

Edit Cells

Cut currently selected cells to clipboard

Paste cells from clipboard above current cell

Paste cells from clipboard on top of current cell

Revert "Delete Cells" invocation

Merge current cell with the one above

Move current cell up

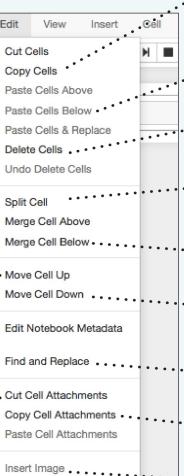
Adjust metadata underlying the current notebook

Remove cell attachments

Paste attachments of current cell

Insert Cells

Add new cell above the current one



Copy cells from clipboard to current cursor position

Paste cells from clipboard below current cell

Delete current cells

Split up a cell from current cursor position

Merge current cell with the one below

Move current cell down

Find and replace in selected cells

Copy attachments of current cell

Insert image in selected cells

Working with Different Programming Languages

Kernels provide computation and communication with front-end interfaces like the notebooks. There are three main kernels:



IPython



IRkernel



Julia

Installing Jupyter Notebook will automatically install the IPython kernel.

Restart kernel

Restart kernel & run all cells

Restart kernel & run all cells

Kernel

Widgets

Help

Interrupt

Restart

Restart & Clear Output

Restart & Run All

Reconnect

Shutdown

Change kernel

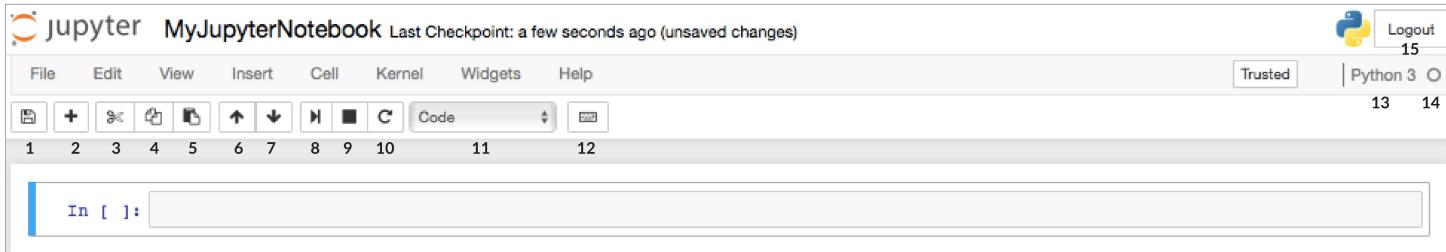
Interrupt kernel

Interrupt kernel & clear all output

Connect back to a remote notebook

Run other installed kernels

Command Mode:



Edit Mode:



Executing Cells

Run selected cell(s)

Run current cells down and create a new one above

Run all cells above the current cell

Change the cell type of current cell

toggle, toggle scrolling and clear all output

Cell

Kernel

Widgets

Run current cells down and create a new one below

Run all cells

Run all cells below the current cell

toggle, toggle scrolling and clear current outputs

Run Cells

Run Cells and Select Below

Run Cells and Insert Below

Run All

Run All Above

Run All Below

Cell Type

Current Outputs

All Output

Toggle Header

Toggle Toolbar

Toggle Line Numbers

Cell Toolbar

Toggle line numbers in cells

Toggle display of Jupyter logo and filename

Toggle display of toolbar

Toggle display of cell action icons:

- None

- Edit metadata

- Raw cell format

- Slideshow

- Attachments

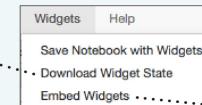
- Tags

Widgets

Notebook widgets provide the ability to visualize and control changes in your data, often as a control like a slider, textbox, etc.

You can use them to build interactive GUIs for your notebooks or to synchronize stateful and stateless information between Python and JavaScript.

Download serialized state of all widget models in use

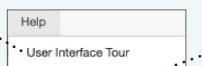


Save notebook with interactive widgets
Embed current widgets

1. Save and checkpoint
2. Insert cell below
3. Cut cell
4. Copy cell(s)
5. Paste cell(s) below
6. Move cell up
7. Move cell down
8. Run current cell
9. Interrupt kernel
10. Restart kernel
11. Display characteristics
12. Open command palette
13. Current kernel
14. Kernel status
15. Log out from notebook server

Asking For Help

Walk through a UI tour



List of built-in keyboard shortcuts

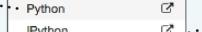
Notebook help topics

Description of markdown available in notebook



Information on unofficial Jupyter Notebook extensions

Python help topics



IPython help topics

NumPy help topics



SciPy help topics

Matplotlib help topics



SymPy help topics

Pandas help topics



About Jupyter Notebook

DataCamp

Learn Python for Data Science Interactively



Python For Data Science Cheat Sheet

NumPy Basics

Learn Python for Data Science Interactively at www.DataCamp.com



NumPy

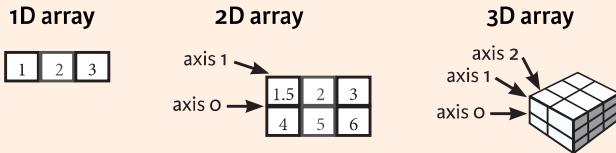
The NumPy library is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

Use the following import convention:

```
>>> import numpy as np
```



NumPy Arrays



Creating Arrays

```
>>> a = np.array([1,2,3])
>>> b = np.array([(1.5,2,3), (4,5,6)], dtype = float)
>>> c = np.array([(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]),
      dtype = float)
```

Initial Placeholders

<pre>>>> np.zeros((3,4))</pre>	Create an array of zeros
<pre>>>> np.ones((2,3),dtype=np.int16)</pre>	Create an array of ones
<pre>>>> d = np.arange(10,25,5)</pre>	Create an array of evenly spaced values (step value)
<pre>>>> np.linspace(0,2,9)</pre>	Create an array of evenly spaced values (number of samples)
<pre>>>> e = np.full((2,2),7)</pre>	Create a constant array
<pre>>>> f = np.eye(2)</pre>	Create a 2X2 identity matrix
<pre>>>> np.random.random((2,2))</pre>	Create an array with random values
<pre>>>> np.empty((3,2))</pre>	Create an empty array

I/O

Saving & Loading On Disk

```
>>> np.save('my_array', a)
>>> np.savetxt('array.npy', a, b)
>>> np.load('my_array.npy')
```

Saving & Loading Text Files

```
>>> np.loadtxt("myfile.txt")
>>> np.genfromtxt("myfile.csv", delimiter=',')
>>> np.savetxt("myarray.txt", a, delimiter=" ")
```

Data Types

<pre>>>> np.int64</pre>	Signed 64-bit integer types
<pre>>>> np.float32</pre>	Standard double-precision floating point
<pre>>>> np.complex</pre>	Complex numbers represented by 128 floats
<pre>>>> np.bool</pre>	Boolean type storing TRUE and FALSE values
<pre>>>> np.object</pre>	Python object type
<pre>>>> np.string_</pre>	Fixed-length string type
<pre>>>> np_unicode_</pre>	Fixed-length unicode type

Inspecting Your Array

```
>>> a.shape
>>> len(a)
>>> b.ndim
>>> e.size
>>> b.dtype
>>> b.dtype.name
>>> b.astype(int)
```

Array dimensions
Length of array
Number of array dimensions
Number of array elements
Data type of array elements
Name of data type
Convert an array to a different type

Asking For Help

```
>>> np.info(np.ndarray.dtype)
```

Array Mathematics

Arithmetic Operations

<pre>>>> g = a - b array([-0.5, 0. , 0.], [-3. , -3. , -3.])</pre>	Subtraction
<pre>>>> np.subtract(a,b)</pre>	Subtraction
<pre>>>> b + a array([[2.5, 4. , 6.], [5. , 7. , 9.]])</pre>	Addition
<pre>>>> np.add(b,a)</pre>	Addition
<pre>>>> a / b array([[0.66666667, 1. , [0.25, 0.4, 0.5]],</pre>	Division
<pre>>>> np.divide(a,b)</pre>	Division
<pre>>>> a * b array([[1.5, 4. , 9.], [4. , 10. , 18.]])</pre>	Multiplication
<pre>>>> np.multiply(a,b)</pre>	Multiplication
<pre>>>> np.exp(b)</pre>	Exponentiation
<pre>>>> np.sqrt(b)</pre>	Square root
<pre>>>> np.sin(a)</pre>	Print sines of an array
<pre>>>> np.cos(b)</pre>	Element-wise cosine
<pre>>>> np.log(a)</pre>	Element-wise natural logarithm
<pre>>>> e.dot(f) array([[7. , 7.], [7. , 7.]])</pre>	Dot product

Comparison

<pre>>>> a == b array([[False, True, True], [False, False, False]], dtype=bool)</pre>	Element-wise comparison
<pre>>>> a < 2 array([True, False, False], dtype=bool)</pre>	Element-wise comparison
<pre>>>> np.array_equal(a, b)</pre>	Array-wise comparison

Aggregate Functions

<pre>>>> a.sum()</pre>	Array-wise sum
<pre>>>> a.min()</pre>	Array-wise minimum value
<pre>>>> b.max(axis=0)</pre>	Maximum value of an array row
<pre>>>> b.cumsum(axis=1)</pre>	Cumulative sum of the elements
<pre>>>> a.mean()</pre>	Mean
<pre>>>> b.median()</pre>	Median
<pre>>>> a.corrcoef()</pre>	Correlation coefficient
<pre>>>> np.std(b)</pre>	Standard deviation

Copying Arrays

```
>>> h = a.view()
>>> np.copy(a)
>>> h = a.copy()
```

Create a view of the array with the same data
Create a copy of the array
Create a deep copy of the array

Sorting Arrays

```
>>> a.sort()
>>> c.sort(axis=0)
```

Sort an array
Sort the elements of an array's axis

Subsetting, Slicing, Indexing

Subsetting

```
>>> a[2]
3
>>> b[1,2]
6.0
```

Select the element at the 2nd index
Select the element at row 0 column 2 (equivalent to b[1][2])

Slicing

```
>>> a[0:2]
array([1, 2])
>>> b[0:2,1]
array([ 2.,  5.])
```

Select items at index 0 and 1
Select items at rows 0 and 1 in column 1

```
>>> b[:1]
array([[1.5, 2., 3.]])
```

Select all items at row 0 (equivalent to b[0:1, :])
Same as [1, :, :]

```
>>> c[1,:]
array([[ 3.,  2.,  1.],
       [ 4.,  5.,  6.]])
```

Reversed array a

Boolean Indexing

```
>>> a[a<2]
array([1])
```

Select elements from a less than 2

Fancy Indexing

```
>>> b[[1, 0, 1, 0], [0, 1, 2, 0]]
array([ 4.,  2.,  6.,  1.5])
>>> b[[1, 0, 1, 0]][:, [0, 1, 2, 0]]
array([[ 4.,  5.,  6.,  4.],
       [ 1.5,  2.,  3.,  1.5],
       [ 4.,  5.,  6.,  4.],
       [ 1.5,  2.,  3.,  1.5]])
```

Select elements (1,0),(0,1),(1,2) and (0,0)
Select a subset of the matrix's rows and columns

Array Manipulation

Transposing Array

```
>>> i = np.transpose(b)
>>> i.T
```

Permute array dimensions
Permute array dimensions

Changing Array Shape

```
>>> b.ravel()
>>> g.reshape(3,-2)
```

Flatten the array
Reshape, but don't change data

Adding/Removing Elements

```
>>> h.resize((2,6))
>>> np.append(h,g)
>>> np.insert(a, 1, 5)
>>> np.delete(a, [1])
```

Return a new array with shape (2,6)
Append items to an array
Insert items in an array
Delete items from an array

Combining Arrays

```
>>> np.concatenate((a,d),axis=0)
array([ 1,  2,  3, 10, 15, 20])
>>> np.vstack((a,b))
array([[ 1.,  2.,  3.],
       [ 1.5,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> np.r_[e,f]
>>> np.hstack((e,f))
array([[ 7.,  7.,  1.,  0.],
       [ 7.,  7.,  0.,  1.]])
>>> np.column_stack((a,d))
array([[ 1, 10],
       [ 2, 15],
       [ 3, 20]])
>>> np.c_[a,d]
```

Concatenate arrays
Stack arrays vertically (row-wise)
Stack arrays vertically (row-wise)
Stack arrays horizontally (column-wise)
Create stacked column-wise arrays
Create stacked column-wise arrays

Splitting Arrays

```
>>> np.hsplit(a,3)
[array([1]),array([2]),array([3])]
>>> np.vsplit(c,2)
[array([[ 1.5,  2.,  3.],
       [ 4.,  5.,  6.]]),
 array([[ 3.,  2.,  3.],
       [ 4.,  5.,  6.]])]
```

Split the array horizontally at the 3rd index
Split the array vertically at the 2nd index



Python For Data Science Cheat Sheet

Also see NumPy

SciPy - Linear Algebra

Learn More Python for Data Science [Interactively](#) at www.datacamp.com



SciPy

The SciPy library is one of the core packages for scientific computing that provides mathematical algorithms and convenience functions built on the NumPy extension of Python.



Interacting With NumPy

[Also see NumPy](#)

```
>>> import numpy as np  
>>> a = np.array([1,2,3])  
>>> b = np.array([(1+5j),2j,3j], [4j,5j,6j])  
>>> c = np.array([(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)])
```

Index Tricks

>>> np.mgrid[0:5,0:5] >>> np.ogrid[0:2,0:2] >>> np.r_[3,0]*5,-1:1:10j >>> np.c_[b,c]	Create a dense meshgrid Create an open meshgrid Stack arrays vertically (row-wise) Create stacked column-wise arrays
---	---

Shape Manipulation

>>> np.transpose(b) >>> b.flatten() >>> np.hstack((b,c)) >>> np.vstack((a,b)) >>> np.hsplit(c,2) >>> np.vsplit(d,2)	Permute array dimensions Flatten the array Stack arrays horizontally (column-wise) Stack arrays vertically (row-wise) Split the array horizontally at the 2nd index Split the array vertically at the 2nd index
--	--

Polynomials

```
>>> from numpy import poly1d  
>>> p = poly1d([3,4,5])
```

Create a polynomial object

Vectorizing Functions

```
>>> def myfunc(a):  
    if a < 0:  
        return a**2  
    else:  
        return a/2  
>>> np.vectorize(myfunc)
```

Vectorize functions

Type Handling

```
>>> np.real(c)  
>>> np.imag(c)  
>>> np.real_if_close(c,tol=1000)  
>>> np.cast['f'](np.pi)
```

Return the real part of the array elements
Return the imaginary part of the array elements
Return a real array if complex parts close to 0
Cast object to a data type

Other Useful Functions

```
>>> np.angle(b,deg=True)  
>>> g = np.linspace(0,np.pi,num=5)  
>>> g[3:] += np.pi  
>>> np.unwrap(g)  
>>> np.logspace(0,10,3)  
>>> np.select([c<4],[c*2])  
  
>>> misc.factorial(a)  
>>> misc.comb(10,3,exact=True)  
>>> misc.central_diff_weights(3)  
>>> misc.derivative(myfunc,1.0)
```

Return the angle of the complex argument
Create an array of evenly spaced values
(number of samples)
Unwrap
Create an array of evenly spaced values (log scale)
Return values from a list of arrays depending on conditions
Factorial
Combine N things taken at k time
Weights for N-point central derivative
Find the n-th derivative of a function at a point

Linear Algebra

You'll use the `linalg` and `sparse` modules. Note that `scipy.linalg` contains and expands on `numpy.linalg`.

```
>>> from scipy import linalg, sparse
```

Creating Matrices

```
>>> A = np.matrix(np.random.random((2,2)))  
>>> B = np.asmatrix(b)  
>>> C = np.mat(np.random.random((10,5)))  
>>> D = np.mat([[3,4], [5,6]])
```

Basic Matrix Routines

Inverse

```
>>> A.I  
>>> linalg.inv(A)  
>>> A.T  
>>> A.H  
>>> np.trace(A)
```

Norm

```
>>> linalg.norm(A)  
>>> linalg.norm(A,1)  
>>> linalg.norm(A,np.inf)
```

Rank

```
>>> np.linalg.matrix_rank(C)
```

Determinant

```
>>> linalg.det(A)
```

Solving linear problems

```
>>> linalg.solve(A,b)  
>>> E = np.mat(a).T  
>>> linalg.lstsq(D,E)
```

Generalized inverse

```
>>> linalg.pinv(C)  
>>> linalg.pinv2(C)
```

Creating Sparse Matrices

```
>>> F = np.eye(3, k=1)  
>>> G = np.mat(np.identity(2))  
>>> C[C > 0.5] = 0  
>>> H = sparse.csr_matrix(C)  
>>> I = sparse.csc_matrix(D)  
>>> J = sparse.dok_matrix(A)  
>>> E.todense()  
>>> sparse.isspmatrix_csc(A)
```

Create a 2x2 identity matrix
Create a 2x2 identity matrix
Compressed Sparse Row matrix
Compressed Sparse Column matrix
Dictionary Of Keys matrix
Sparse matrix to full matrix
Identify sparse matrix

Sparse Matrix Routines

Inverse

```
>>> sparse.linalg.inv(I)
```

Norm

```
>>> sparse.linalg.norm(I)
```

Solving linear problems

```
>>> sparse.linalg.spsolve(H,I)
```

Sparse Matrix Functions

```
>>> sparse.linalg.expm(I)      Sparse matrix exponential
```

Matrix Functions

Addition

```
>>> np.add(A,D)
```

Subtraction

```
>>> np.subtract(A,D)
```

Division

```
>>> np.divide(A,D)
```

Multiplication

```
>>> np.multiply(D,A)  
>>> np.dot(A,D)  
>>> np.vdot(A,D)  
>>> np.inner(A,D)  
>>> np.outer(A,D)  
>>> np.tensordot(A,D)  
>>> np.kron(A,D)
```

Exponential Functions

```
>>> linalg.expm(A)  
>>> linalg.expm2(A)  
>>> linalg.expm3(D)
```

Logarithm Function

```
>>> linalg.logm(A)
```

Trigonometric Functions

```
>>> linalg.sinm(D)  
>>> linalg.cosm(D)  
>>> linalg.tanm(A)
```

Hyperbolic Trigonometric Functions

```
>>> linalg.sinhm(D)  
>>> linalg.coshm(D)  
>>> linalg.tanhm(A)
```

Matrix Sign Function

```
>>> np.signm(A)
```

Matrix Square Root

```
>>> linalg.sqrtm(A)
```

Arbitrary Functions

```
>>> linalg.funm(A, lambda x: x*x)
```

Addition

Subtraction

Division

Multiplication

Dot product

Vector dot product

Inner product

Outer product

Tensor dot product

Kronecker product

Matrix exponential

Matrix exponential (Taylor Series)

Matrix exponential (eigenvalue decomposition)

Matrix logarithm

Matrix sine

Matrix cosine

Matrix tangent

Hypberbolic matrix sine

Hyperbolic matrix cosine

Hyperbolic matrix tangent

Matrix sign function

Matrix square root

Evaluate matrix function

Decompositions

Eigenvalues and Eigenvectors

```
>>> la, v = linalg.eig(A)
```

Solve ordinary or generalized eigenvalue problem for square matrix
Unpack eigenvalues
First eigenvector
Second eigenvector
Unpack eigenvalues

Singular Value Decomposition

```
>>> U,s,Vh = linalg.svd(B)  
>>> M,N = B.shape  
>>> Sig = linalg.diagsvd(s,M,N)
```

Singular Value Decomposition (SVD)
Construct sigma matrix in SVD

LU Decomposition

```
>>> P,L,U = linalg.lu(C)
```

LU Decomposition

Sparse Matrix Decompositions

```
>>> la, v = sparse.linalg.eigs(F,1)  
>>> sparse.linalg.svds(H, 2)
```

Eigenvalues and eigenvectors
SVD

Asking For Help

```
>>> help(scipy.linalg.diagsvd)  
>>> np.info(np.matrix)
```

DataCamp

Learn Python for Data Science [Interactively](#)



Python For Data Science Cheat Sheet

Pandas Basics

Learn Python for Data Science Interactively at www.DataCamp.com



Pandas

The Pandas library is built on NumPy and provides easy-to-use data structures and data analysis tools for the Python programming language.



Use the following import convention:

```
>>> import pandas as pd
```

Pandas Data Structures

Series

A one-dimensional labeled array capable of holding any data type

a	3
b	-5
c	7
d	4

```
>>> s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
```

DataFrame

Columns

	Country	Capital	Population
0	Belgium	Brussels	11190846
1	India	New Delhi	1303171035
2	Brazil	Brasilia	207847528

```
>>> data = {'Country': ['Belgium', 'India', 'Brazil'],
   >>>          'Capital': ['Brussels', 'New Delhi', 'Brasilia'],
   >>>          'Population': [11190846, 1303171035, 207847528]}
>>> df = pd.DataFrame(data,
   >>>                  columns=['Country', 'Capital', 'Population'])
```

I/O

Read and Write to CSV

```
>>> pd.read_csv('file.csv', header=None, nrows=5)
>>> df.to_csv('myDataFrame.csv')
```

Read and Write to Excel

```
>>> pd.read_excel('file.xlsx')
>>> pd.to_excel('dir/myDataFrame.xlsx', sheet_name='Sheet1')
Read multiple sheets from the same file
>>> xlsx = pd.ExcelFile('file.xlsx')
>>> df = pd.read_excel(xlsx, 'Sheet1')
```

Asking For Help

```
>>> help(pd.Series.loc)
```

Selection

Getting

```
>>> s['b']
-5
>>> df[1:]
   Country    Capital  Population
1  India      New Delhi     1303171035
2  Brazil     Brasilia     207847528
```

Also see NumPy Arrays

Get one element

Get subset of a DataFrame

Selecting, Boolean Indexing & Setting

By Position

```
>>> df.iloc[0, [0]]
'Belgium'
>>> df.iat[0, [0]]
'Belgium'
```

By Label

```
>>> df.loc[[0], ['Country']]
'Belgium'
>>> df.at[[0], ['Country']]
'Belgium'
```

By Label/Position

```
>>> df.ix[2]
   Country      Brazil
   Capital    Brasilia
   Population  207847528
```

```
>>> df.ix[:, 'Capital']
0    Brussels
1  New Delhi
2    Brasilia
```

```
>>> df.ix[1, 'Capital']
'New Delhi'
```

Boolean Indexing

```
>>> s[s > 1]
>>> s[(s < -1) | (s > 2)]
>>> df[df['Population'] > 1200000000]
```

Setting

```
>>> s['a'] = 6
```

Select single value by row & column

Select single value by row & column labels

Select single row or subset of rows

Select a single column of subset of columns

Select rows and columns

Series s where value is not >1

s where value is <-1 or >2

Use filter to adjust DataFrame

Set index a of Series s to 6

Dropping

```
>>> s.drop(['a', 'c'])
>>> df.drop('Country', axis=1)
```

Drop values from rows (axis=0)

Drop values from columns(axis=1)

Sort & Rank

```
>>> df.sort_index()
>>> df.sort_values(by='Country')
>>> df.rank()
```

Sort by labels along an axis

Sort by the values along an axis

Assign ranks to entries

Retrieving Series/DataFrame Information

Basic Information

```
>>> df.shape
>>> df.index
>>> df.columns
>>> df.info()
>>> df.count()
```

(rows,columns)
Describe index
Describe DataFrame columns
Info on DataFrame
Number of non-NA values

Summary

```
>>> df.sum()
>>> df.cumsum()
>>> df.min()/df.max()
>>> df.idxmin()/df.idxmax()
>>> df.describe()
>>> df.mean()
>>> df.median()
```

Sum of values
Cummulative sum of values
Minimum/maximum values
Minimum/Maximum index value
Summary statistics
Mean of values
Median of values

Applying Functions

```
>>> f = lambda x: x*x
>>> df.apply(f)
>>> df.applymap(f)
```

Apply function
Apply function element-wise

Data Alignment

Internal Data Alignment

NA values are introduced in the indices that don't overlap:

```
>>> s3 = pd.Series([7, -2, 3], index=['a', 'c', 'd'])
>>> s + s3
a    10.0
b    NaN
c     5.0
d     7.0
```

Arithmetic Operations with Fill Methods

You can also do the internal data alignment yourself with the help of the fill methods:

```
>>> s.add(s3, fill_value=0)
a    10.0
b   -5.0
c     5.0
d     7.0
>>> s.sub(s3, fill_value=2)
>>> s.div(s3, fill_value=4)
>>> s.mul(s3, fill_value=3)
```



Python For Data Science Cheat Sheet

Scikit-Learn

Learn Python for data science interactively at www.DataCamp.com



Scikit-learn

Scikit-learn is an open source Python library that implements a range of machine learning, preprocessing, cross-validation and visualization algorithms using a unified interface.

A Basic Example

```
>>> from sklearn import neighbors, datasets, preprocessing
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.metrics import accuracy_score
>>> iris = datasets.load_iris()
>>> X, y = iris.data[:, :2], iris.target
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=33)
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> X_train = scaler.transform(X_train)
>>> X_test = scaler.transform(X_test)
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
>>> knn.fit(X_train, y_train)
>>> y_pred = knn.predict(X_test)
>>> accuracy_score(y_test, y_pred)
```

Loading The Data

Also see NumPy & Pandas

Your data needs to be numeric and stored as NumPy arrays or SciPy sparse matrices. Other types that are convertible to numeric arrays, such as Pandas DataFrame, are also acceptable.

```
>>> import numpy as np
>>> X = np.random.random((10, 5))
>>> y = np.array(['M', 'M', 'F', 'F', 'M', 'F', 'M', 'F', 'F'])
>>> X[X < 0.7] = 0
```

Training And Test Data

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(X,
...                                                    y,
...                                                    random_state=0)
```

Preprocessing The Data

Standardization

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler().fit(X_train)
>>> standardized_X = scaler.transform(X_train)
>>> standardized_X_test = scaler.transform(X_test)
```

Normalization

```
>>> from sklearn.preprocessing import Normalizer
>>> scaler = Normalizer().fit(X_train)
>>> normalized_X = scaler.transform(X_train)
>>> normalized_X_test = scaler.transform(X_test)
```

Binarization

```
>>> from sklearn.preprocessing import Binarizer
>>> binarizer = Binarizer(threshold=0.0).fit(X)
>>> binary_X = binarizer.transform(X)
```

Create Your Model

Supervised Learning Estimators

Linear Regression

```
>>> from sklearn.linear_model import LinearRegression
>>> lr = LinearRegression(normalize=True)
```

Support Vector Machines (SVM)

```
>>> from sklearn.svm import SVC
>>> svc = SVC(kernel='linear')
```

Naive Bayes

```
>>> from sklearn.naive_bayes import GaussianNB
>>> gnb = GaussianNB()
```

KNN

```
>>> from sklearn import neighbors
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
```

Unsupervised Learning Estimators

Principal Component Analysis (PCA)

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=0.95)
```

K Means

```
>>> from sklearn.cluster import KMeans
>>> k_means = KMeans(n_clusters=3, random_state=0)
```

Model Fitting

Supervised learning

```
>>> lr.fit(X, y)
>>> knn.fit(X_train, y_train)
>>> svc.fit(X_train, y_train)
```

Unsupervised Learning

```
>>> k_means.fit(X_train)
>>> pca_model = pca.fit_transform(X_train)
```

Fit the model to the data

Fit the model to the data

Fit to data, then transform it

Prediction

Supervised Estimators

```
>>> y_pred = svc.predict(np.random.random((2,5)))
>>> y_pred = lr.predict(X_test)
>>> y_pred = knn.predict_proba(X_test)
```

Unsupervised Estimators

```
>>> y_pred = k_means.predict(X_test)
```

Predict labels

Predict labels

Estimate probability of a label

Predict labels in clustering algos

Evaluate Your Model's Performance

Classification Metrics

Accuracy Score

```
>>> knn.score(X_test, y_test)
>>> from sklearn.metrics import accuracy_score
>>> accuracy_score(y_test, y_pred)
```

Estimator score method

Metric scoring functions

Classification Report

```
>>> from sklearn.metrics import classification_report
>>> print(classification_report(y_test, y_pred))
```

Precision, recall, f1-score and support

Confusion Matrix

```
>>> from sklearn.metrics import confusion_matrix
>>> print(confusion_matrix(y_test, y_pred))
```

Regression Metrics

Mean Absolute Error

```
>>> from sklearn.metrics import mean_absolute_error
>>> y_true = [3, -0.5, 2]
>>> mean_absolute_error(y_true, y_pred)
```

Mean Squared Error

```
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(y_test, y_pred)
```

R² Score

```
>>> from sklearn.metrics import r2_score
>>> r2_score(y_true, y_pred)
```

Clustering Metrics

Adjusted Rand Index

```
>>> from sklearn.metrics import adjusted_rand_score
>>> adjusted_rand_score(y_true, y_pred)
```

Homogeneity

```
>>> from sklearn.metrics import homogeneity_score
>>> homogeneity_score(y_true, y_pred)
```

V-measure

```
>>> from sklearn.metrics import v_measure_score
>>> metrics.v_measure_score(y_true, y_pred)
```

Cross-Validation

```
>>> from sklearn.cross_validation import cross_val_score
>>> print(cross_val_score(knn, X_train, y_train, cv=4))
>>> print(cross_val_score(lr, X, y, cv=2))
```

Tune Your Model

Grid Search

```
>>> from sklearn.grid_search import GridSearchCV
>>> params = {"n_neighbors": np.arange(1,3),
...            "metric": ["euclidean", "cityblock"]}
>>> grid = GridSearchCV(estimator=knn,
...                      param_grid=params)
>>> grid.fit(X_train, y_train)
>>> print(grid.best_score_)
>>> print(grid.best_estimator_.n_neighbors)
```

Randomized Parameter Optimization

```
>>> from sklearn.grid_search import RandomizedSearchCV
>>> params = {"n_neighbors": range(1,5),
...            "weights": ["uniform", "distance"]}
>>> rsearch = RandomizedSearchCV(estimator=knk,
...                               param_distributions=params,
...                               cv=4,
...                               n_iter=8,
...                               random_state=5)
>>> rsearch.fit(X_train, y_train)
>>> print(rsearch.best_score_)
```



Python For Data Science Cheat Sheet

Matplotlib

Learn Python Interactively at www.DataCamp.com



Matplotlib

Matplotlib is a Python 2D plotting library which produces publication-quality figures in a variety of hardcopy formats and interactive environments across platforms.



1) Prepare The Data

Also see [Lists & NumPy](#)

1D Data

```
>>> import numpy as np  
>>> x = np.linspace(0, 10, 100)  
>>> y = np.cos(x)  
>>> z = np.sin(x)
```

2D Data or Images

```
>>> data = 2 * np.random.random((10, 10))  
>>> data2 = 3 * np.random.random((10, 10))  
>>> Y, X = np.mgrid[-3:3:100j, -3:3:100j]  
>>> U = -1 - X**2 + Y  
>>> V = 1 + X - Y**2  
>>> from matplotlib.cbook import get_sample_data  
>>> img = np.load(get_sample_data('axes_grid/bivariate_normal.npy'))
```

2) Create Plot

```
>>> import matplotlib.pyplot as plt
```

Figure

```
>>> fig = plt.figure()  
>>> fig2 = plt.figure(figsize=plt.figaspect(2.0))
```

Axes

All plotting is done with respect to an Axes. In most cases, a subplot will fit your needs. A subplot is an axes on a grid system.

```
>>> fig.add_axes()  
>>> ax1 = fig.add_subplot(221) # row-col-num  
>>> ax3 = fig.add_subplot(212)  
>>> fig3, axes = plt.subplots(nrows=2, ncols=2)  
>>> fig4, axes2 = plt.subplots(ncols=3)
```

3) Plotting Routines

1D Data

```
>>> fig, ax = plt.subplots()  
>>> lines = ax.plot(x, y)  
>>> ax.scatter(x, y)  
>>> axes[0,0].bar([1,2,3],[3,4,5])  
>>> axes[1,0].barh([0.5,1.25],[0,1,2])  
>>> axes[1,1].axhline(0.45)  
>>> axes[0,1].axvline(0.65)  
>>> ax.fill(x,y,color='blue')  
>>> ax.fill_between(x,y,color='yellow')
```

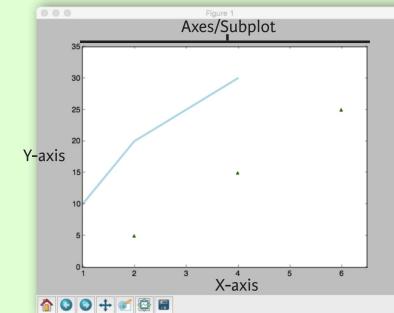
2D Data or Images

```
>>> fig, ax = plt.subplots()  
>>> im = ax.imshow(img,  
                  cmap='gist_earth',  
                  interpolation='nearest',  
                  vmin=-2,  
                  vmax=2)
```

Colormapped or RGB arrays

Plot Anatomy & Workflow

Plot Anatomy



Figure

Workflow

The basic steps to creating plots with matplotlib are:

- 1 Prepare data
- 2 Create plot
- 3 Plot
- 4 Customize plot
- 5 Save plot
- 6 Show plot

```
>>> import matplotlib.pyplot as plt  
>>> x = [1,2,3,4]  
>>> y = [10,20,25,30] Step 1  
>>> fig = plt.figure() Step 2  
>>> ax = fig.add_subplot(111) Step 3  
>>> ax.plot(x, y, color='lightblue', linewidth=3) Step 3,4  
>>> ax.scatter([2,4,6],  
             [5,15,25],  
             color='darkgreen',  
             marker='^')  
>>> ax.set_xlim(1, 6.5)  
>>> plt.savefig('foo.png')  
>>> plt.show() Step 6
```

4) Customize Plot

Colors, Color Bars & Color Maps

```
>>> plt.plot(x, x, x, x**2, x, x**3)  
>>> ax.plot(x, y, alpha = 0.4)  
>>> ax.plot(x, y, c='k')  
>>> fig.colorbar(im, orientation='horizontal')  
>>> im = ax.imshow(img,  
                  cmap='seismic')
```

Markers

```
>>> fig, ax = plt.subplots()  
>>> ax.scatter(x,y,marker=".")  
>>> ax.plot(x,y,marker="o")
```

Linestyles

```
>>> plt.plot(x,y,linewidth=4.0)  
>>> plt.plot(x,y,ls='solid')  
>>> plt.plot(x,y,ls='--')  
>>> plt.plot(x,y,'--',x**2,y**2,'-.')  
>>> plt.setp(lines,color='r',linewidth=4.0)
```

Text & Annotations

```
>>> ax.text(1,-2.1,  
           'Example Graph',  
           style='italic')  
>>> ax.annotate("Sine",  
               xy=(8, 0),  
               xycoords='data',  
               xytext=(10.5, 0),  
               textcoords='data',  
               arrowprops=dict(arrowstyle="->",  
                               connectionstyle="arc3"),)
```

Vector Fields

```
>>> axes[0,1].arrow(0,0,0.5,0.5)  
>>> axes[1,1].quiver(y,z)  
>>> axes[0,1].streamplot(X,Y,U,V)
```

Mathtext

```
>>> plt.title(r'$\sigma_i=15$', fontsize=20)
```

Limits, Legends & Layouts

```
>>> ax.margins(x=0.0,y=0.1)  
>>> ax.axis('equal')  
>>> ax.set(xlim=[0,10.5], ylim=[-1.5,1.5])  
>>> ax.set_xlim(0,10.5)
```

Legends

```
>>> ax.set(title='An Example Axes',  
           ylabel='Y-Axis',  
           xlabel='X-Axis')  
>>> ax.legend(loc='best')
```

Ticks

```
>>> ax.xaxis.set(ticks=range(1,5),  
                ticklabels=[3,100,-12,"foo"])  
>>> ax.tick_params(axis='y',  
                           direction='inout',  
                           length=10)
```

Subplot Spacing

```
>>> fig3.subplots_adjust(wspace=0.5,  
                        hspace=0.3,  
                        left=0.125,  
                        right=0.9,  
                        top=0.9,  
                        bottom=0.1)  
>>> fig.tight_layout()
```

Axis Spines

```
>>> ax1.spines['top'].set_visible(False)  
>>> ax1.spines['bottom'].set_position(('outward',10))
```

Add padding to a plot
Set the aspect ratio of the plot to 1
Set limits for x-and y-axis
Set limits for x-axis

Set a title and x-and y-axis labels

No overlapping plot elements

Manually set x-ticks

Make y-ticks longer and go in and out

Adjust the spacing between subplots

Fit subplot(s) in to the figure area

Make the top axis line for a plot invisible
Move the bottom axis line outward

5) Save Plot

Save figures

```
>>> plt.savefig('foo.png')
```

Save transparent figures

```
>>> plt.savefig('foo.png', transparent=True)
```

6) Show Plot

```
>>> plt.show()
```

Close & Clear

```
>>> plt.clf()
```

```
>>> plt.close()
```

Clear an axis
Clear the entire figure
Close a window



Python For Data Science Cheat Sheet

Seaborn

Learn Data Science **Interactively** at www.DataCamp.com



Statistical Data Visualization With Seaborn

The Python visualization library **Seaborn** is based on **matplotlib** and provides a high-level interface for drawing attractive statistical graphics.

Make use of the following aliases to import the libraries:

```
>>> import matplotlib.pyplot as plt  
>>> import seaborn as sns
```

The basic steps to creating plots with Seaborn are:

1. Prepare some data
2. Control figure aesthetics
3. Plot with Seaborn
4. Further customize your plot

```
>>> import matplotlib.pyplot as plt  
>>> import seaborn as sns  
>>> tips = sns.load_dataset("tips")  
Step 1  
>>> sns.set_style("whitegrid")  
Step 2  
>>> g = sns.lmplot(x="tip",  
y="total_bill",  
data=tips,  
aspect=2)  
  
>>> g = (g.set_axis_labels("Tip", "Total bill(USD)")  
set(xlim=(0,10), ylim=(0,100)))  
Step 3  
>>> plt.title("title")  
Step 4  
>>> plt.show(g) Step 5
```

1) Data

Also see Lists, NumPy & Pandas

```
>>> import pandas as pd  
>>> import numpy as np  
>>> uniform_data = np.random.rand(10, 12)  
>>> data = pd.DataFrame({'x':np.arange(1,101),  
y':np.random.normal(0,4,100)})
```

Seaborn also offers built-in data sets:

```
>>> titanic = sns.load_dataset("titanic")  
>>> iris = sns.load_dataset("iris")
```

2) Figure Aesthetics

```
>>> f, ax = plt.subplots(figsize=(5, 6)) Create a figure and one subplot
```

Seaborn styles

```
>>> sns.set()  
>>> sns.set_style("whitegrid")  
>>> sns.set_style("ticks",  
{"xtick.major.size":8,  
"ytick.major.size":8})  
>>> sns.axes_style("whitegrid")
```

(Re)set the seaborn default
Set the matplotlib parameters
Set the matplotlib parameters
Return a dict of params or use with
with to temporarily set the style

3) Plotting With Seaborn

Axis Grids

```
>>> g = sns.FacetGrid(titanic,  
col="survived",  
row="sex")  
>>> g.map(plt.hist, "age")  
>>> sns.factorplot(x="pclass",  
y="survived",  
hue="sex",  
data=titanic)  
>>> sns.lmplot(x="sepal_width",  
y="sepal_length",  
hue="species",  
data=iris)
```

Subplot grid for plotting conditional relationships

Draw a categorical plot onto a Facetgrid

Plot data and regression model fits across a FacetGrid

```
>>> h = sns.PairGrid(iris)  
>>> h = h.map(plt.scatter)  
>>> sns.pairplot(iris)  
>>> i = sns.JointGrid(x="x",  
y="y",  
data=data)  
>>> i = i.plot(sns.regplot,  
sns.distplot)  
>>> sns.jointplot("sepal_length",  
"sepal_width",  
data=iris,  
kind='kde')
```

Subplot grid for plotting pairwise relationships
Plot pairwise bivariate distributions
Grid for bivariate plot with marginal univariate plots

Plot bivariate distribution

Categorical Plots

Scatterplot

```
>>> sns.stripplot(x="species",  
y="petal_length",  
data=iris)  
>>> sns.swarmplot(x="species",  
y="petal_length",  
data=iris)
```

Bar Chart

```
>>> sns.barplot(x="sex",  
y="survived",  
hue="class",  
data=titanic)
```

Count Plot

```
>>> sns.countplot(x="deck",  
data=titanic,  
palette="Greens_d")
```

Point Plot

```
>>> sns.pointplot(x="class",  
y="survived",  
hue="sex",  
data=titanic,  
palette={"male":"g",  
"female":"m"},  
markers=["^", "o"],  
linestyles=[ "--", "-."])
```

Boxplot

```
>>> sns.boxplot(x="alive",  
y="age",  
hue="adult_male",  
data=titanic)
```

Violinplot

```
>>> sns.violinplot(x="age",  
y="sex",  
hue="survived",  
data=titanic)
```

Scatterplot with one categorical variable

Categorical scatterplot with non-overlapping points

Show point estimates and confidence intervals with scatterplot glyphs

Show count of observations

Show point estimates and confidence intervals as rectangular bars

Boxplot

Boxplot with wide-form data

Violin plot

Subplot grid for plotting pairwise relationships
Plot pairwise bivariate distributions
Grid for bivariate plot with marginal univariate plots

Plot bivariate distribution

Regression Plots

```
>>> sns.regplot(x="sepal_width",  
y="sepal_length",  
data=iris,  
ax=ax)
```

Plot data and a linear regression model fit

Distribution Plots

```
>>> plot = sns.distplot(data.y,  
kde=False,  
color="b")
```

Plot univariate distribution

Matrix Plots

```
>>> sns.heatmap(uniform_data, vmin=0, vmax=1) Heatmap
```

4) Further Customizations

Also see Matplotlib

Axisgrid Objects

```
>>> g.despine(left=True)  
>>> g.set_ylabels("Survived")  
>>> g.set_xticklabels(rotation=45)  
>>> g.set_axis_labels("Survived",  
"Sex")  
>>> h.set(xlim=(0,5),  
ylim=(0,5),  
xticks=[0,2.5,5],  
yticks=[0,2.5,5])
```

Remove left spine
Set the labels of the y-axis
Set the tick labels for x
Set the axis labels

Set the limit and ticks of the x-and y-axis

Plot

```
>>> plt.title("A Title")  
>>> plt.ylabel("Survived")  
>>> plt.xlabel("Sex")  
>>> plt.ylim(0,100)  
>>> plt.xlim(0,10)  
>>> plt.setp(ax,yticks=[0,5])  
>>> plt.tight_layout()
```

Add plot title
Adjust the label of the y-axis
Adjust the label of the x-axis
Adjust the limits of the y-axis
Adjust the limits of the x-axis
Adjust a plot property
Adjust subplot params

5) Show or Save Plot

Also see Matplotlib

```
>>> plt.show()  
>>> plt.savefig("foo.png")  
>>> plt.savefig("foo.png",  
transparent=True)
```

Show the plot
Save the plot as a figure
Save transparent figure

Close & Clear

```
>>> plt.cla()  
>>> plt.clf()  
>>> plt.close()
```

Clear an axis
Clear an entire figure
Close a window



Python For Data Science Cheat Sheet

Bokeh

Learn Bokeh **Interactively** at www.DataCamp.com, taught by Bryan Van de Ven, core contributor

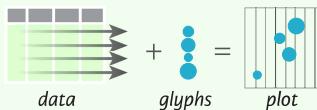


Plotting With Bokeh

The Python interactive visualization library **Bokeh** enables high-performance visual presentation of large datasets in modern web browsers.



Bokeh's mid-level general purpose `bokeh.plotting` interface is centered around two main components: data and glyphs.



The basic steps to creating plots with the `bokeh.plotting` interface are:

1. Prepare some data:
Python lists, NumPy arrays, Pandas DataFrames and other sequences of values
2. Create a new plot
3. Add renderers for your data, with visual customizations
4. Specify where to generate the output
5. Show or save the results

```
>>> from bokeh.plotting import figure
>>> from bokeh.io import output_file, show
>>> x = [1, 2, 3, 4, 5]           Step 1
>>> y = [6, 7, 2, 4, 5]
>>> p = figure(title="simple line example",      Step 2
              x_axis_label='x',
              y_axis_label='y')
>>> p.line(x, y, legend="Temp.", line_width=2)    Step 3
>>> output_file("lines.html")                    Step 4
>>> show(p)                                     Step 5
```

1) Data

Also see Lists, NumPy & Pandas

Under the hood, your data is converted to Column Data Sources. You can also do this manually:

```
>>> import numpy as np
>>> import pandas as pd
>>> df = pd.DataFrame(np.array([[33.9, 4, 65, 'US'],
                                [32.4, 4, 66, 'Asia'],
                                [21.4, 4, 109, 'Europe']]),
                     columns=['mpg', 'cyl', 'hp', 'origin'],
                     index=['Toyota', 'Fiat', 'Volvo'])

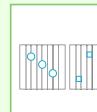
>>> from bokeh.models import ColumnDataSource
>>> cds_df = ColumnDataSource(df)
```

2) Plotting

```
>>> from bokeh.plotting import figure
>>> p1 = figure(plot_width=300, tools='pan,box_zoom')
>>> p2 = figure(plot_width=300, plot_height=300,
               x_range=(0, 8), y_range=(0, 8))
>>> p3 = figure()
```

3) Renderers & Visual Customizations

Glyphs



Scatter Markers

```
>>> p1.circle(np.array([1,2,3]), np.array([3,2,1]),
             fill_color='white')
>>> p2.square(np.array([1.5,3.5,5.5]), [1,4,3],
             color='blue', size=1)
```



Line Glyphs

```
>>> p1.line([1,2,3,4], [3,4,5,6], line_width=2)
>>> p2.multi_line(pd.DataFrame([[1,2,3],[5,6,7]]),
                  pd.DataFrame([[3,4,5],[3,2,1]]),
                  color="blue")
```

Customized Glyphs

Also see Data



Selection and Non-Selection Glyphs

```
>>> p = figure(tools='box_select')
>>> p.circle('mpg', 'cyl', source=cds_df,
             selection_color='red',
             nonselection_alpha=0.1)
```



Hover Glyphs

```
>>> from bokeh.models import HoverTool
>>> hover = HoverTool(tooltips=None, mode='vline')
>>> p3.add_tools(hover)
```



Colormapping

```
>>> from bokeh.models import CategoricalColorMapper
>>> color_mapper = CategoricalColorMapper(
                factors=['US', 'Asia', 'Europe'],
                palette=['blue', 'red', 'green'])
>>> p3.circle('mpg', 'cyl', source=cds_df,
             color=dict(field='origin',
                        transform=color_mapper),
             legend='Origin')
```

Legend Location

Inside Plot Area

```
>>> p.legend.location = 'bottom_left'
```

Outside Plot Area

```
>>> from bokeh.models import Legend
>>> r1 = p2.asterisk(np.array([1,2,3]), np.array([3,2,1]))
>>> r2 = p2.line([1,2,3,4], [3,4,5,6])
>>> legend = Legend(items=[("One", [p1, r1]), ("Two", [r2])],
                     location=(0, -30))
>>> p.add_layout(legend, 'right')
```

Legend Orientation

```
>>> p.legend.orientation = "horizontal"
>>> p.legend.orientation = "vertical"
```

Legend Background & Border

```
>>> p.legend.border_line_color = "navy"
>>> p.legend.background_fill_color = "white"
```

Rows & Columns Layout

Rows

```
>>> from bokeh.layouts import row
>>> layout = row(p1,p2,p3)
```

Columns

```
>>> from bokeh.layouts import column
>>> layout = column(p1,p2,p3)
```

Nesting Rows & Columns

```
>>> layout = row(column(p1,p2), p3)
```

Grid Layout

```
>>> from bokeh.layouts import gridplot
>>> row1 = [p1,p2]
>>> row2 = [p3]
>>> layout = gridplot([[p1,p2], [p3]])
```

Tabbed Layout

```
>>> from bokeh.models.widgets import Panel, Tabs
>>> tab1 = Panel(child=p1, title="tab1")
>>> tab2 = Panel(child=p2, title="tab2")
>>> layout = Tabs(tabs=[tab1, tab2])
```

Linked Plots

Linked Axes

```
>>> p2.x_range = p1.x_range
>>> p2.y_range = p1.y_range
```

Linked Brushing

```
>>> p4 = figure(plot_width = 100,
               tools='box_select,lasso_select')
>>> p4.circle('mpg', 'cyl', source=cds_df)
>>> p5 = figure(plot_width = 200,
               tools='box_select,lasso_select')
>>> p5.circle('mpg', 'hp', source=cds_df)
>>> layout = row(p4,p5)
```

4) Output & Export

Notebook

```
>>> from bokeh.io import output_notebook, show
>>> output_notebook()
```

HTML

Standalone HTML

```
>>> from bokeh.embed import file_html
>>> from bokeh.resources import CDN
>>> html = file_html(p, CDN, "my_plot")
```

```
>>> from bokeh.io import output_file, show
>>> output_file('my_bar_chart.html', mode='cdn')
```

Components

```
>>> from bokeh.embed import components
>>> script, div = components(p)
```

PNG

```
>>> from bokeh.io import export_png
>>> export_png(p, filename="plot.png")
```

SVG

```
>>> from bokeh.io import export_svgs
>>> p.output_backend = "svg"
>>> export_svgs(p, filename="plot.svg")
```

5) Show or Save Your Plots

```
>>> show(p1)
>>> save(p1)
```

```
>>> show(layout)
>>> save(layout)
```

